AD-787 218

EMULATING A HONEYWELL 6180 COMPUTER SYSTEM

Edmund L. Burke, et al

Mitre Corporation

Prepared for:

Rome Air Development Center

June 1974

EMULATING A HONEYWELL 6180 COMPUTER SYSTEM

Edmund L. Burke
Morrie Gasser
W. Lee Schiller

MITRE Corporation

FOREWORD

This interim technical report describes research accomplished between 30 September and 31 December 1973, by MITRE Corporation, Bedford, Massachusetts, under Contract F19628-73-C-0001, Job Order 55500802. Rome Air Development Center, Griffiss Air Force Base, New York, was the controlling government agency; Major William W. Patterson (ISCA) the RADC Project Engineer.

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved.

APPROVED: William W Patterson

WILLIAM W. PATTERSON, Maj, USAF
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Asst Ch, Information Sciences Division

FOR THE COMMANDER: Carlo P. Crocetti

CARLO P. CROCETTI
Chief, Plans Office

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-74-137	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 757 215
4. TITLE *(and Subtitle)* EMULATING A HONEYWELL 6180 COMPUTER SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Technical Report Interim 30 Sep - 31 Dec 73
		6. PERFORMING ORG. REPORT NUMBER MTR-2742
7. AUTHOR(s) Edmund L. Burke Morrie Gasser W. Lee Schiller		8. CONTRACT OR GRANT NUMBER(s) F19628-73-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS MITRE Corporation Dept D72 Bedford, MA 01730		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS JO 55500802
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCA) Griffiss Air Force Base, New York 13441		12. REPORT DATE June 1974
		13. NUMBER OF PAGES 79
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* Same		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release. Distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Microprogramming
Emulation
Honeywell 6180
Computers
Burroughs D-Machine
Nanodata QM-1
Burroughs B1700

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The Honeywell 6180 is a new, large-scale computer for the Multics time-sharing system. This report describes the 6180, and examines the feasibility of emulating it with each of three microprogrammable processors: the Burroughs D-Machine, the Nanodata QM-1, and the Burroughs B1700. Benchmark emulations are presented for each of these machines.

ABSTRACT

The Honeywell 6180 is a new large-scale computer for the Multics timesharing system. This report describes the 6180, and examines the feasibility of emulating it with each of three microprogrammable processors: the Burroughs D-Machine, the Nanodata QM-1, and the Burroughs B1700. Benchmark emulations are presented for each of these machines.

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I

INTRODUCTION

This report presents results of a study of the feasibility of emulating a very sophisticated large scale computer system, the Honeywell 6180, on three microprogrammable processors: the Burroughs D-machine, the Burroughs 1700 and the Nanodata QM-1. The 6180 is a third generation computer that supports the current version of the Multics system. Multics is a large timesharing system that embodies such state-of-the-art concepts as virtual memory, dynamic (execution time) linking, and advanced access controls and presents one of the most challenging architectures to an emulator of any existing system. The microprogrammable machines picked for the study were chosen after a cursory evaluation of available hardware indicated that these were the only machines with the potential for the emulation. As new microprogrammable processors become available, the emulation methodology developed in this report should make the evaluation of these new processors straightforward.

This report assumes the reader is familiar with some of the basic features of Multics. Chapter I of The Multics System by Organick [14] should be considered a prerequisite for readers of this report. In addition, "The Multics Virtual Memory" by Bensoussan et al [10] will be very helpful. Although both these publications discuss the implementation of Multics on the GE645 (a precursor of the 6180), most of the details have not been changed.

Four sections follow this introduction. Section II presents the 6180 architecture in some detail. Owing to the paucity of documentation for the 6180 at the time of this writing, this information was gleaned from a wide variety of sources. Section III develops a methodology for the emulation of the 6180. The significantly greater complexity of the Multics hardware as compared to other third generation machines requires that new ideas be developed for implementing the 6180 operations in microprograms. Section IV then takes a typical 6180 operation and examines the emulation of that operation on the three microprogrammed processors. This examination is in sufficient detail to allow a benchmark to be established for each emulation. Finally, Section V summarizes the major findings and conclusions of the report.

SECTION II

THE ARCHITECTURE OF THE HONEYWELL 6180 SYSTEM

INTRODUCTION

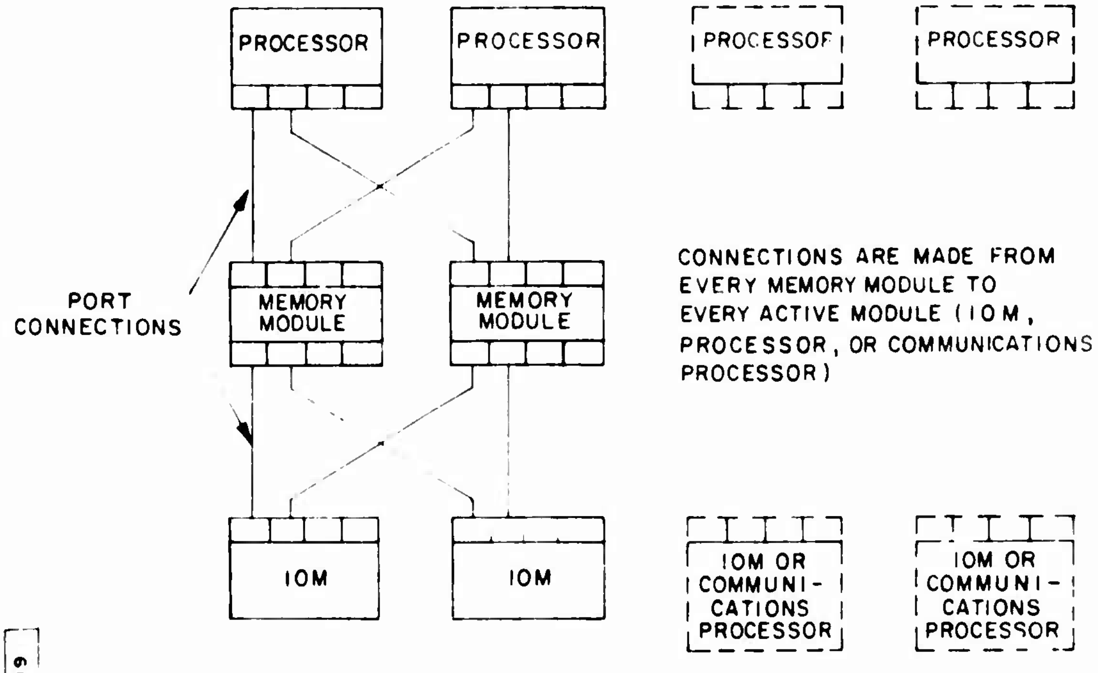
This section describes the architecture of the 6180 or Multics computer system. The descriptive material is directed toward the intricacies of the instruction execution cycle and dynamic address translation (DAT) algorithm and is complete enough to allow the development of an emulation methodology and benchmark.

The Honeywell 6180 computer system supports the current version of the Multics operating system and is synonymous with Multics. The 6180 has evolved from the current generation of Honeywell hardware, the 6000 series. The 6180 is essentially the Honeywell 6080 modified with DAT hardware for the Multics virtual memory. In fact, a switch on the 6180 allows it to run as a 6080. The 6080 is, in turn, the largest and most sophisticated processor in the 6000 series, embodying an extended instruction set, EIS, (for character string and decimal arithmetic processing) and the fastest circuitry and memory of the series.

The 6180 Multics system was first offered as a product in second quarter 1973 and only one 6180 Multics is presently (September 1973) available to the general user. This available Multics is at MIT. Because the system is so recent, standard documentation does not exist. Thus the following description comes from a wide variety of sources that commonly provide preliminary documentation for evolutionary systems. These sources include Honeywell 645 processor (the original Multics machine), 6000 series and EIS manuals, MIT Informations Processing Center Memos, and articles in the Communications of the ACM. Because of the diverse origin of the descriptive materials the presentation of the 6180 architecture should not be construed as final.

SYSTEM DESCRIPTION

The Multics system is composed of processor, memory and input/output modules as well as a front-end processor as shown in Figure 1. While the 6180 processor is the active element in the system, and as such deserves pre-eminent treatment in a discussion of the Multics architecture, it is necessary to have sufficient understanding of all system modules, their interconnections and their functional characteristics to appreciate processor operations.



CONNECTIONS ARE MADE FROM EVERY MEMORY MODULE TO EVERY ACTIVE MODULE (IOM, PROCESSOR, OR COMMUNICATIONS PROCESSOR)

1A-41,909

Figure 1. A Typical Configuration of the Multics System

The number of each module type in the 6180 system shown in Figure 1 is not fixed. A representative example, the MIT installation, has two 6180 processor modules, three memory modules, one IOM (Input/Output Multiplexor), one communications processor (A Datanet 355), and a two million word bulk store subsystem (replacing the second generation paging drum). Each memory module provides storage for 128K (K=1024) 36-bit words. The operation of modules other than the processor module is quite conventional. The bulk store facility provides high speed secondary storage for the virtual memory and looks like an IOM with a high speed I/O device. The bulk store, therefore, is functionally equivalent to an IOM.

SYSTEM PERFORMANCE

The objective of the benchmark emulation is to provide some comparative measure of the time to perform some function on the 6180 vs the time to perform the same function on each of the emulators. This measure implies that the time to perform certain functions on the 6180 is well established. It is not. Sufficient data exists, however, on the previous Multics performance and on the 6080 to allow the extrapolation of 6180 performance data.

Since the 6180 and 6080 differ only in the DAT hardware, if functions are chosen so that the DAT algorithm is not executed, then 6080 timings can be used to predict the 6180 timings. Table 1 shows the execution times, in microseconds, for four representative 6080 instructions. The STA instruction is used as the common function for

Instruction Type	6070/6080 Timing[1] (μsecs)
Load A (LDA)	0.7
Store A (STA)	1.0
No operation (NOP)	0.7
Floating add (FAD)	1.7

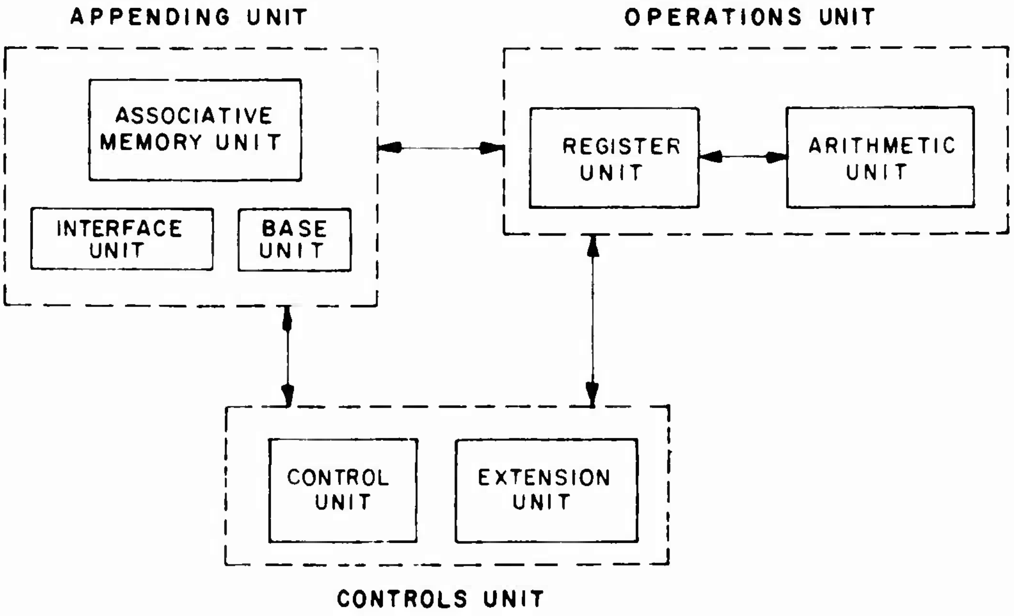
Table I. Instruction Execution Times

(1) Taken from HIS manual [7]

comparisons of execution times between the 6180 and the various emulators (see section IV).

PROCESSOR CONFIGURATION

A block diagram of the processor module is shown below in Figure 2. The processor consists of three functional boxes: the



IA-41,897

Figure 2. Block Diagram of the Processor Module

appending unit, the operations unit and the controls unit. The three functions can be briefly described as follows:

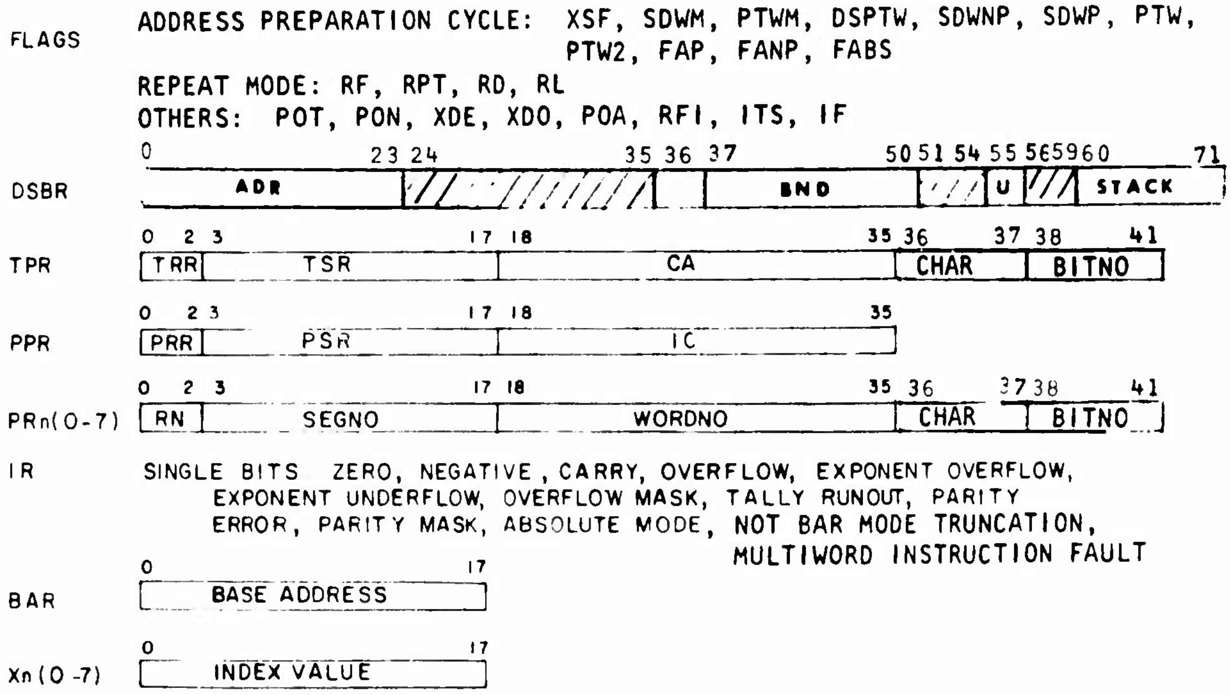
> Controls Unit: This unit performs instruction fetch, decode, execution, and effective address calculation which may require multiple indirect word fetches. The control unit portion operates on the standard instruction set and the extension unit operates on the extended instruction set. The extended instruction set is a feature of the even numbered 6000-series processors (i.e., 6040, 6060, 6080) and includes instruction decode and address calculation of its own.
>
> Operations Unit: This unit performs the arithmetic and logic operations specified by the various instructions, and loads and unloads certain registers.
>
> Appending Unit: This unit performs the various operations needed to implement the DAT (Dynamic Address Translation) algorithm. An instruction can address memory in the virtual (append) mode or the absolute mode. When addressing in the virtual mode, the appending unit takes the two-dimensional address and converts it to the appropriate 24-bit absolute address.

Before discussing the operation of each of the above units in detail, it is appropriate to look at the registers which define each unit and discuss how each register is used. Due to the lack of documentation at this level, it is difficult to develop a taxonomy for the registers that associates each register with a functional box. Thus the following groupings of registers with module functions may not be entirely accurate.

Controls Unit Registers

The registers functionally belonging to the Controls Unit are shown in Figure 3. These registers are briefly described as follows:

FLAGS: This is a series of one or more bit registers that indicate the state in the very complex cycle between instruction fetch and eventual instruction execution.


FLAGS
ADDRESS PREPARATION CYCLE: XSF, SDWM, PTWM, DSPTW, SDWNP, SDWP, PTW, PTW2, FAP, FANP, FABS
REPEAT MODE: RF, RPT, RD, RL
OTHERS: POT, PON, XDE, XDO, POA, RFI, ITS, IF
DSBR
0 23 24 35 36 37 50 51 54 55 56 59 60 71
ADR
BND
U
STACK
TPR
0 2 3 17 18 35 36 37 38 41
TRR
TSR
CA
CHAR
BITNO
PPR
0 2 3 17 18 35
PRR
PSR
IC
PRn(0-7)
0 2 3 17 18 35 36 37 38 41
RN
SEGNO
WORDNO
CHAR
BITNO
IR
SINGLE BITS ZERO, NEGATIVE, CARRY, OVERFLOW, EXPONENT OVERFLOW, EXPONENT UNDERFLOW, OVERFLOW MASK, TALLY RUNOUT, PARITY ERROR, PARITY MASK, ABSOLUTE MODE, NOT BAR MODE TRUNCATION, MULTIWORD INSTRUCTION FAULT
BAR
0 17
BASE ADDRESS
Xn(0-7)
0 17
INDEX VALUE

IA-41,898

Figure 3. Controls Unit Registers

DSBR: Descriptor Segment Base Register. This register contains the absolute address of the descriptor segment for the current process (DSBR.ADR), specifies the size of the segment (DSBR.BND), whether or not it is paged (DSBR.U), and where the stack can be found (DSBR.STACK). More detailed discussion of the use of this register will be found in the section on address preparation.

TPR: Temporary Pointer Register. Virtual addresses are stored in pointer registers. The TPR is used as a general scratch register to hold the virtual address that would correspond to the memory address register in conventional machines. TPR.TSR holds the segment number, TPR.CA holds the current address of the word in the segment, and TPR.TRR is the ring number associated with the current reference to that segment. PR.CHAR and PR.BITNO are character and bit addresses used by the Extended Instruction Set.

PPR: Processor Pointer Register. This register functions as a virtual instruction counter and is structured similarly to TPR. PPR.IC is the actual 18-bit instruction counter register. PPR.P specifies that this is a privileged procedure.

PRn(0-7): There are 8 pointer registers directly addressable by the programmer. They are used to store indirect addresses to segments that are frequently referenced. Their use avoids the extra time required to make an indirect reference through core. The fields are functionally the same as the corresponding fields in TPR and PPR.

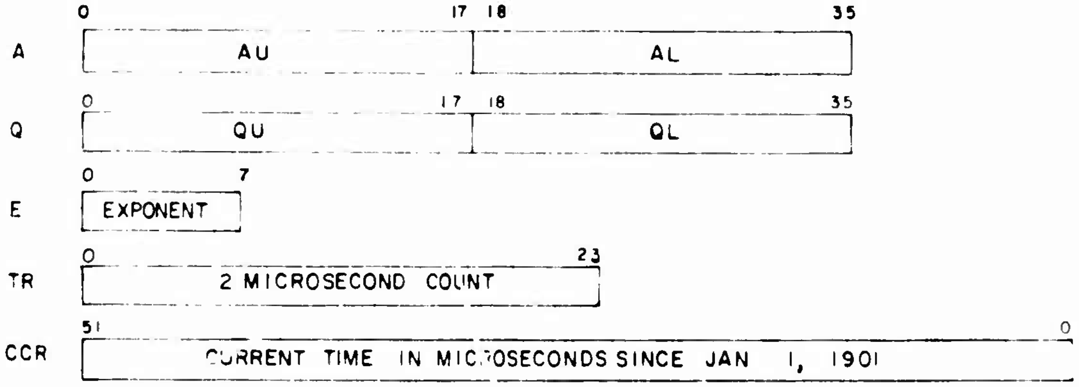
IR: Indicator Register. This register contains the indicator bits, error flags, and mode bits for various conditions found on conventional machines.

BAR: Base Address Register. For downward compatability, the 6180 can run as a 6080 by entering the BAR mode. In this mode all addresses within a segment are one dimensional addresses relative to the value in this register.

Xn(0-7): Index Registers. These 8 registers are general 18-bit registers that can be used for indexing in a conventional manner.

Operations Unit Registers

The Operations Unit contains the three programmer working registers: accumulator (A), quotient (Q), and exponent (E); and two registers for timing: timer register (TR) to provide a relative time base and calendar clock (CCR) that contains a current time.



[1A.41,899]

Figure 4. Operations Unit Registers

Appending Unit Registers

The Appending Unit is used to implement the DAT algorithm. There are no registers that can be associated with the Appending Unit as such, however the unit deals with two data structures in core that it must also store internally. The segment descriptor word (SDW), which is actually a double word, and the page table word (PTW) are illustrated in Figure 5.

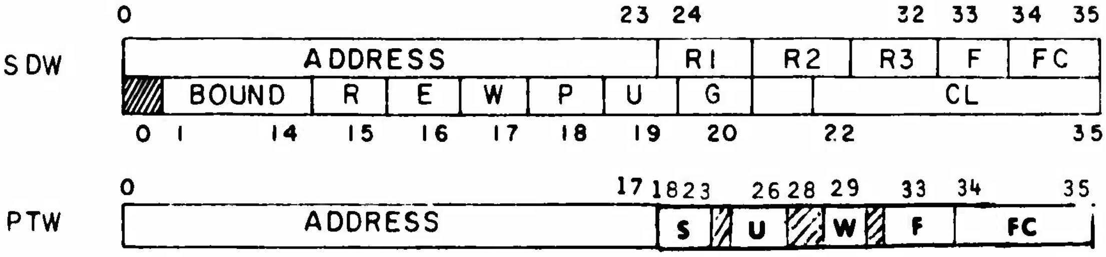


Figure 5. Appending Unit Registers

SDW: This double word contains vital information concerning the segment currently in use.

ADDRESS: Absolute core address (mod 64) of page table for segment, or address of segment if not paged.

R1, R2, R3: Ring numbers (0-7) determining read, write, execute, and call brackets as follows:

Read bracket is 0 - R2.
Write bracket is 0 - R1.
Execute bracket is R1. - R2.
Call bracket is (R2+1) - R3.

F: If off, a directed fault is to take place when this segment is referenced.

FC: If F is off, the fault code contained in this field is used.

BOUND: Highest 16-word block of segment that can be addressed.

R, E, W: Read, execute and write permit indicators. If on, access is permitted provided ring bracket is satisfied.

P: If on and processor is executing in ring zero, the privileged mode is entered.

U: If on, segment is unpaged.

G: Gate indicator. If off, calls to this segment must be directed to an address less than CL.

CL: Call limiter, above which calls to this segment may not be made.

PTW: This word contains information about the current page being accessed.

ADDRESS: Absolute address of this page in memory.
S: For use by software
U: If on, page has been used.

W: If on, page has been written into.

F, FC: Same as for SDW.

The Appending Unit also contains two associative memories: one holds the last 16 SDW's and the other holds the last 16 PTW's that were referenced. Along with all the information from the SDW or PTW, the segment and page numbers associated with these are saved so that an SDW or PTW can be identified by an associative search.

PROCESSOR OPERATION

This subsection discusses the operation of the processor in four parts. The first two discuss address preparation, the instruction cycle, and interrupts in general. The last two parts discuss the same subjects in more detail.

Address Preparation -- General

Address preparation for an operand is defined to be the calculation of a two-dimensional address (segment number, offset) that addresses the operand in virtual memory. Conversion of this virtual address to a physical address is referred to as DAT. In the 6180, DAT can be thought of as a subroutine that is called whenever a memory reference is made in the append mode of operation. Address preparation involves the indexing and fetching of indirect words necessary to determine the offset and sometimes the segment number of the operand referenced by an instruction. Each memory reference made through DAT requires reference to the segment descriptor word and page table word for the word being referenced, thus making it possible for various faults and access violations to occur any time DAT is used. Since words are always fetched in even-odd pairs, which must of course both be in the same page, DAT need only operate with the even part of the offset.

DAT should be thought of only as an address translation, separate from the memory access to the location referenced. The reason for excluding the memory access from DAT is that DAT is invoked (at least conceptually) at certain times merely to check on the validity of a virtual address without requiring any reference to the contents of the address. The DAT subroutine has as input two arguments and returns an output or generates a fault. The input arguments are (1) the type of access desired (read, read/write, write, or execute) and (2) the virtual address (segment and offset). The returned value, if no fault is generated, is the 24-bit absolute address. A "fault" is generated when the virtual address is not "in core."

An example of the case where no memory access to the operand is made at the time DAT is invoked is the store instruction. The sequence of events for a store instruction is

(1) calculate address of operand,
(2) call DAT to check on write access,
(3) execute instruction, and
(4) store value into absolute address returned previously by DAT.

The 6180 implements the Multics ring mechanism in hardware. Rings can be considered to be a generalization of the user-supervisor modes, where inner (lower numbered) rings have more privilege than outer (higher numbered) rings. Eight rings are provided, with the most privileged supervisor routines in ring 0 and user routines executing in rings 4-7. A process executing in an inner ring has all the access capabilities it has in an outer ring. Thus when a procedure in an inner ring is called from an outer ring, the current ring of execution is changed to the ring of the inner procedure and access is allowed to all data that the previous (outer) ring had access to, plus new access as required by the inner ring. The ring protection mechanism requires that the current ring of execution be saved in a register, and that each segment descriptor word specify the rings from which access is allowed to that segment.

One of the important aspects of the ring mechanism is that a procedure in an inner ring can assume the more restricted access of an outer ring, if desired, without changing the current ring of execution. The need for this capability is best illustrated in the case where a user routine executing in ring 4, for example, calls a supervisor routine in ring 1. The user supplies arguments pointing to data areas within segments accessible to his own ring (4). The supervisor makes reference to these arguments through pointer registers containing the segment number and offset of the arguments, and the ring number of the user. The effective ring number, computed with each effective address, is the maximum of the ring number in the pointer register and the current ring. This effective ring number, not the current ring number, is used to determine the type of access allowed for the reference. Thus the user cannot force the supervisor to make reference to segments the user didn't have access to in the first place. Moreover, each indirect word pointing to another segment contains a ring number. Thus the user can transmit pointers from arguments passed to him from an outer ring (such as ring 5) to the supervisor. The supervisor, when referencing such an indirect word, will have as its effective ring number the maximum of: supervisor ring (1), user ring (4), and indirect word (5). The user program in ring 4 could of course modify the ring number 5 in the indirect word, but even if it made it less than 4, the pointer register that contains the user ring number would force the access privileges back to those of ring 4.

Instruction Cycle and Interrupts -- General

A machine cycle flowchart for a typical computer would appear as in Figure 6, where interrupts could only occur between instructions. In such a machine only the registers available to software need be saved at an interrupt, and the details of the boxes "prepare operand address" and "execute instruction" do not affect this structure. In the 6180 such a situation cannot be tolerated because the many references ,o DAT r3quired during the operand address preparation phase can generate faults (which are processed just like interrupts). At fault time the internal machine state must be saved in sufficient detail so th it a return from the fault handler can restore the machine to the proper point in the operand address preparation, allowing the process to resume as if the fault had never occurred.

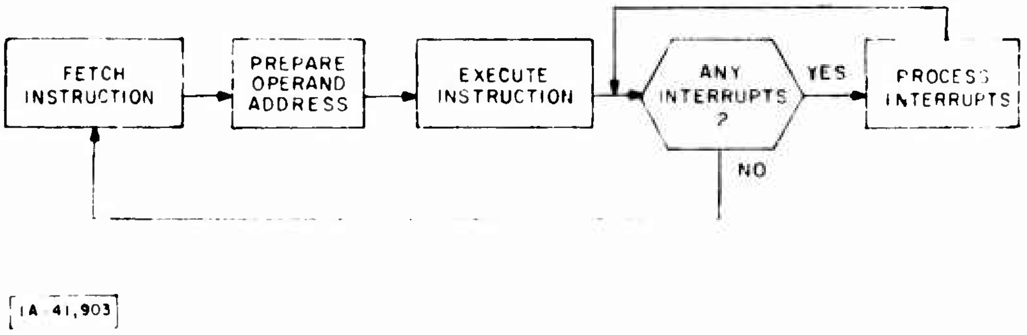
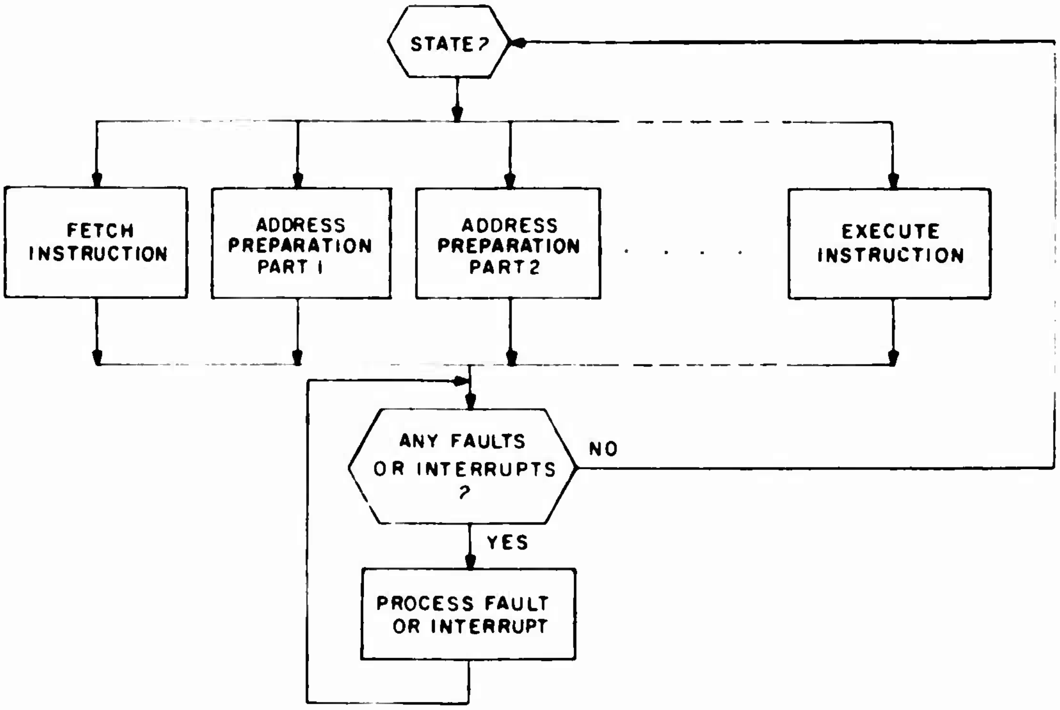


Figure 6. Typical Machine Cycle

Figure 7 shows mcre of what the 6180 instruction cycle is like. The horizontal line of boxes represents different "cycles". Each part of the address preparation that contains a call to DAT must be an independent cycle of the machine. Interrupts and faults are then checl.ed for at defined times between cycles during instruction fetch, operand fetch, and execution. Ncte the distinction between the generation of a fault (which occurs in DAT somewhere in the cycle) and the testing for the fault for the purposes of generating the interrupt after each cycle. This separation is required because other kinds of faults and interrupts could occur from outside the DAT, and,to provide an orderly means for determining priority, interrupts and faults must all be detected at once. It is intentional that only one box was shown for "execute instruction". There appears to be no reason for interrupting the execution of normal (not multi-word) instructions during their execution because all necessary calls to DAT for argument validation have been made previously. The exceptions to this rule might be the transfer and call instructions whose operands need validation by DAT in the execute cycle, but the manner in which these exceptions are handled is not clear from available documentation.


STATE?
FETCH INSTRUCTION
ADDRESS PREPARATION PART 1
ADDRESS PREPARATION PART 2
. . . . .
EXECUTE INSTRUCTION
ANY FAULTS OR INTERRUPTS ?
NO
YES
PROCESS FAULT OR INTERRUPT

IA-41,905

Figure 7. 6180 Instruction Cycle

The box at the top of Figure 7 called "state?" determines which cycle is to be entered next as a function of various internal flags and bits. At an interrupt this state must be saved, along with several internal registers and intermediate results not normally available to software. Software saves this state (reflected in a "snapshot" of the CPU taken at interrupt recognition time) in core with the first instruction of an interrupt or fault routine, and restores this state on return to the interrupted program. Under some conditions it is necessary for software to examine and modify the saved state data.

The data saved on an interrupt by the "store control unit" (SCU) instruction contains 8 words that include both the even and odd numbered instructions currently in execution, the instruction counter, etc. Saving of both even and odd instructions is necessary on the 6180 because the execution of the even instruction takes place at the same time as part of the address preparation for the odd instruction, and memory works in double word widths. Of course, proper handling of faults and interrupts also requires saving of the software accessible registers, but this saving is handled directly by the fault handling software.

Up to this time the EIS multiword instructions have been ignored for the sake of simplicity. The main confusion arising when multiword instructions are considered is that these instructions can handle very long strings that can cross page boundaries, and thus must make DAT references many times during their execution. The EIS processor also has 8 additional words of information that must be saved on an interrupt, by software using the SPL instruction. The EIS processor can probably be represented in Figure 7 by adding one or more "execute EIS instruction" boxes that contain DAT references. Because the EIS instructions go through repetitive loops, the understanding is that these additional boxes may be reentered many times before the instruction is complete.

Address Preparation -- Detail

All non-EIS instructions save a few go through the same address preparation cycles. EIS multiword instructions go through their own address preparations for each of one to three descriptors following the instruction, and most single word EIS instructions have yet another similar but not identical method of address preparation. Most of the following discussion will be exclusively for the non-EIS instructions, with general remarks about differences due to the EIS instructions.

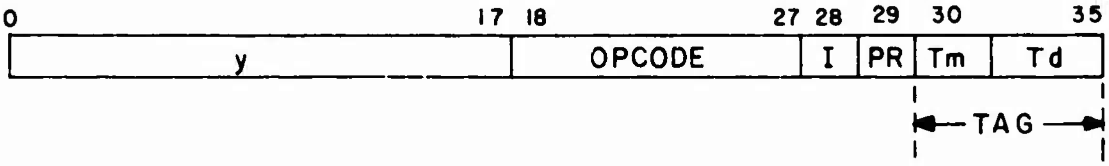


Figure 8. General Instruction Format

All non-EIS instructions have the "general instruction format" as shown in Figure 8. The operand referenced by the instruction is pointed to by the 18-bit address in the "y" field, specifying an offset in the current segment. The TAG field is a complex field specifying what modifications are to be performed on the value "y" to transform it into a final offset "Y" that points to the actual operand. Modifications include indexing and indirection. The first part of this discussion will be concerned with modifications that generate an offset Y in the current segment while the last few modifications described all reference data within another segment.

The tag field, expanded into two subfields, is shown in Figure 9. The major modification category is specified in the Tm field. The Td field specifies either a register (for R, RI, and IR) or more detail on the type of modification in the case of IT modification. A general summary of the various modifications follows, but first it might be helpful to discuss the general format of the indirect word, illustrated in Figure 10. Note that the indirect word format is similar to the instruction format with the opcode missing. Any reference made to a word designated as indirect will be subject to the same modifications as the original instruction, thus permitting any number of levels of indirection. Indirection ceases usually when the tag of the indirect word specifies no further indirection. Note that the tag of an indirect word or instruction usually specifies what modification is to be performed on the y field of the current indirect word, and whether the next word fetched is to be interpreted as an indirect word or not.

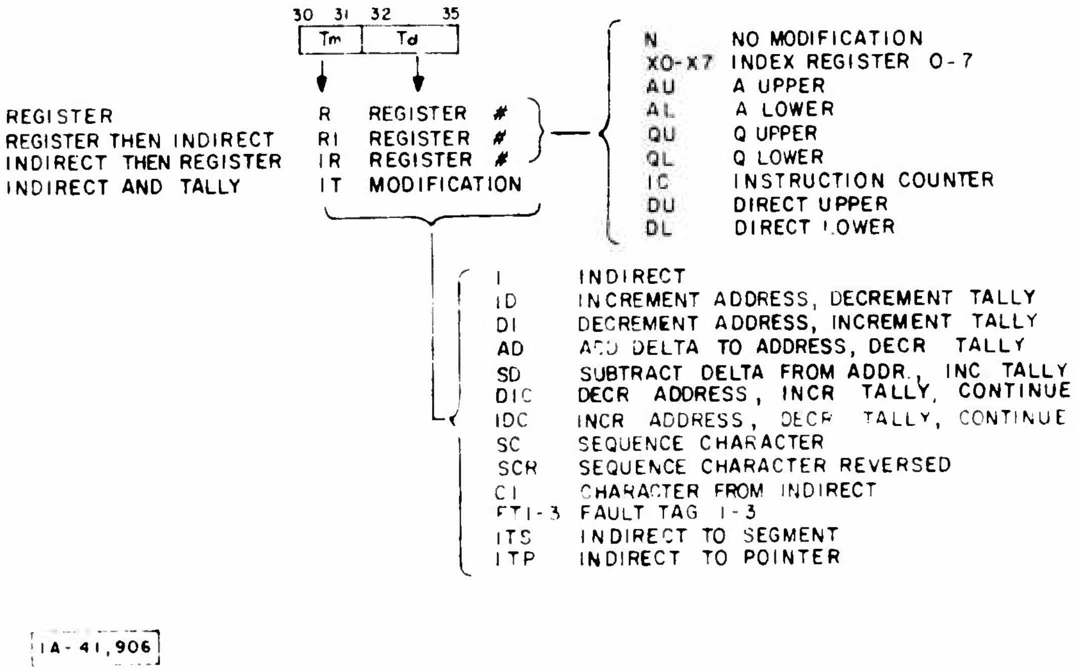


Figure 9. Tag Field

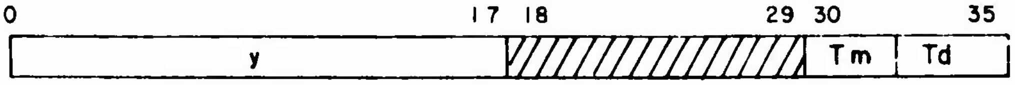


Figure 10. Indirect Word

Address Modification. The four modifications specified in the Td field are summarized as follows:

R Register.
The register specified in the Td field is to be added to the value y to form the 18 bit offset of the operand. No indirection takes place.

RI Register, then indirect.
The register in the Td field is added to y as for R modification, but the result is an offset which points to an indirect word whose y and TAG fields are interpreted further.

IR Indirect, then register.
The y field points directly to an indirect word. The register specified in the current Td field is added to the resultant offset after all further indirection is complete. If one of the subsequent indirect words also specifies IR modification, then the Td field of that word selects the register to be added, and the original Td field is ignored.

IT Indirect and tally.
The y field points to an indirect tally word whose contents is interpreted in a manner determined by the original Td field.

The registers selected by the Tm field for R, RI and IR modifications are described as follows:

N No modifications.
y becomes the offset.

X0-X7 One of 8 index registers.
Xn+y becomes the offset.

AU,AL Upper or lower half of A register.
AU+y or AL+y become the offset.

QU,QL Upper or lower half of Q register.
QU+y or QL+y become the offset.

IC Instruction counter.
IC+y becomes the offset.

DU,DL Direct upper or direct lower.
This specifies that y itself is the operand, rather than the address of the operand. 18 bits of the operand are always zero, and the 18 bits of y become the high or low half of the operand. This modification obviously makes no sense for store instructions.

For the IT (indirect and tally) modification, the Tm field does not specify a register, but can take on one of the following values that determine how the indirect word is to be interpreted.

I Indirect only.
The y field of the indirect word is the address of the operand, and the tag field of the indirect word is ignored. No further indirection occurs.

ID Increment address, decrement tally.

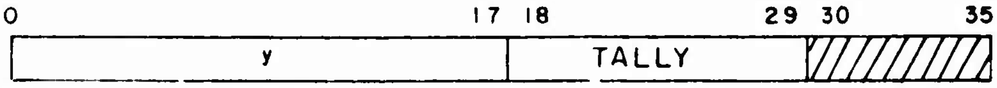


Figure 11. Indirect Word: ID, DI

In this mode the indirect word is interpreted as in Figure 11. The y field of the indirect word points to the operand as for I modification, but after operand fetch the y field is incremented and the tally field is decremented. If the tally becomes zero after decrementing, the tally runout indicator is set.

DI Decrement address, increment tally.
The indirect word is interpreted as for ID (Figure 11). Operation is similar to ID except that the y field is decremented, the tally is incremented, and then the operand is fetched.

IDC Increment address, decrement tally, continue indirection. The indirect word is interpreted as in Figure 12. Operation is identical to ID, except that indirection can continue as specified by the tag field of this indirect word. Note that the tag field was ignored for ID and DI. There is one restriction on the contents of this tag field: If Td specifies R or RI, Tm must specify N (no modification). Td may specify IT or IR, for which there are no restrictions.

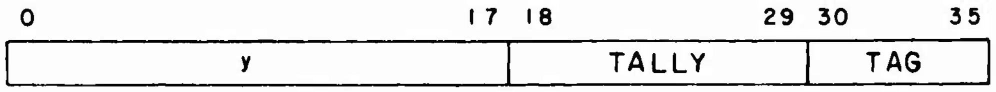


Figure 12. Indirect Tally Word: IDC, DIC

DIC Decrement address, increment tally, continue indirection. This mode is identical to DI, except that indirection can continue as for IDC with similar restrictions.

AD Add delta. Operation is similar to ID, except the y field is incremented by the value of delta contained in the tag field of the indirect tally word. The tally field is still decremented by 1.

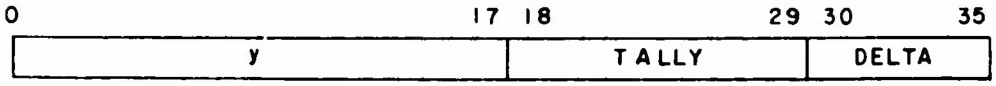


Figure 13. Indirect Tally Word: AD, SD

SD Subtract delta. Similar to DI except that a delta in the indirect word is used to decrement the y field as for AD.

SC Sequence character.
In this mode, the indirect tally word, interpreted as in Figure 14, is used to address a character operand. Bit 30 specifies whether the character is a 6 or 9 bit character, and CHAR is the character number within the word pointed to by y. Operation is similar to ID. Each reference will increment CHAR until it becomes 5 or 3, depending on the character size. On the next reference CHAR will be set to zero and y will be incremented. In this manner sequential characters in a continuous string are referenced.

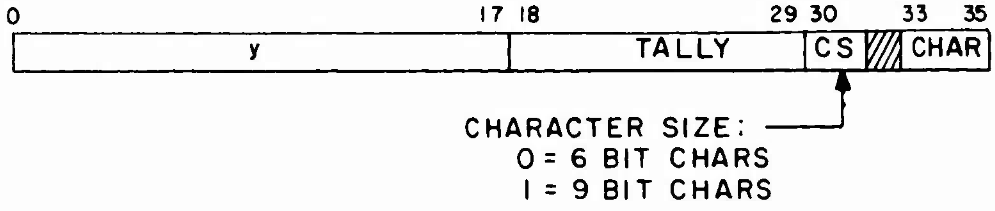


Figure 14. Indirect Tally Word: SC, SCR, CI

SCR Sequence character reversed.
Operation is parallel to DI, where the indirect word is interpreted as for SC. The character count goes backwards until it reaches zero. Then the y field is decremented by one and CHAR is set to 5 or 3.

CI Character from indirect.
The tally word is interpreted as for SC, but on each reference only the tally field is modified, thus permitting repeated references to the same character.

ITS Indirect to segment.
This modification is not a tally word modification. The ITS modifier, appearing in the tag field of an even addressed indirect word, specifies that the indirect word and the following word are used to address a location within another segment. This word pair is shown in Figure 15. The segment number and offset specify the location, and

the tag field of the second word determines what types of normal modifications are to be made to the new offset. The ITS modifier may only appear in an even addressed indirect word, and not in the instruction. The RN field specifies the ring number whose access privileges are to be used when making this reference. The BITNO field is used to address bits for EIS instructions.

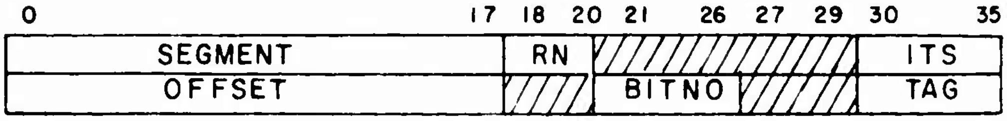


Figure 15. ITS Word Pair

ITP Indirect to pointer.
This modifier is used in a manner similar to ITS, except that the first indirect word specifies a pointer register (PR) instead of a segment number. This pointer register contains a segment number, offset, and ring number, and the offset in the second word is added to the offset in the PR to form the new offset y which becomes the base for further address modification specified in the tag field.

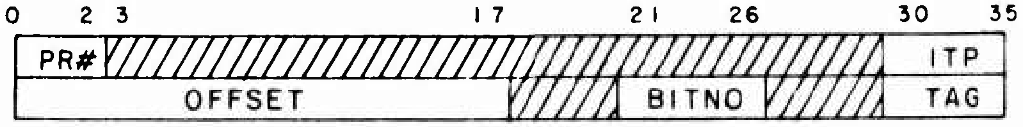


Figure 16. ITP Word Pair

FT1-3 Fault tag 1-3

When this modifier appears in an indirect word or word pair, a directed fault is generated. This fault may be interpreted by software in any way desired, and it is up to software to set this fault code in the indirect word.

One further type of modification, which takes place before any other type of modification, and may only be specified in the original instruction, is a reference through a pointer register (PR). If PR is specified by a bit in the instruction, as in Figure 17, then reference is first made to one of the 8 PRs specified in bits 0-2. The remaining part of the y field, interpreted as a signed 2's complement number, is added to the offset in the PR to determine the initial y value to be used for further modification. The tag field is interpreted as usual. This mode allows reference to another segment without using an indirect (ITS or ITP) word pair.

0 2	3 17	18 27	28	29	30 35
PR#	y	OPCODE	I	1	TAG

Figure 17. PR Modification (bit 29=1)

DAT Details. Figure 18 illustrates how DAT works. As mentioned earlier, the inputs to DAT are the virtual address (segment, offset) and the type of access desired (read, write, or execute). The DAT returns an absolute main memory address of the word referenced. In Figure 18, the TPR contains the virtual address. The TRR field contains a ring number that is initially the current ring of execution (PRR) taken from the PPR, but may later (after a level of indirection to another segment) assume a higher value (and therefore less access) if specified in the indirect word. There is no way for TRR to be less than PRR. Refer to Figure 18 for the following steps.

1. The DSBR points to the first word of the page table for the descriptor segment. This page table contains a word for every page in the descriptor segment.

2. The segment number (TSR) can be thought of as a two part address where the high 5 bits point to the page table word (PTW) and the low 10 bits point to the offset within the page. This step consists of adding the DSBR value to the high part of TSR to obtain the absolute address of the PTW. The PTW contains the address of the page, flags that are set by hardware when the page is written into or referenced, and fault bits set by software that indicate the kind of fault to be generated when this page is referenced. Usually the fault bits are used to signal the fact that the page desired is not in core. If such a fault occurs, DAT exits immediately, setting the appropriate fault indicator for subsequent testing at the end of the current cycle.

3. If no fault is specified, the PTW's address field is used to get to the address of the beginning of the page of the descriptor segment that contains the segment descriptor word (SDW).

4. The low part of TSR is an offset into this page, and the SDW is fetched as a double word. The SDW contains fields describing the segment to be referenced. Besides specifying the address of the page table for the segment, the number of pages in the segment, and faults to be generated if the page table is not in core, it also describes the kind of access the current process has to this segment. The access field allows read, read/write, or execute privilege. The ring fields con'ain ring numbers specifying from which rings each of these accesses are permitted, and a call field specifies at what point entry can be made into this segment via a CALL instruction (which can change to an inner ring of execution). A ring field also specifies the new ring number to be loaded into the PRR when the segment is properly called using the CALL instruction.

5. If no fault is generated from the SDW (either due to an illegal reference or missing segment) then the address of the page table is determined.

6. Assuming 1K pages, the high 9 bits of the offset or current address (CA) are an index into the page table for this segment, and then the PTW is fetched and treated exactly as in step 1, with a possible fault being generated.

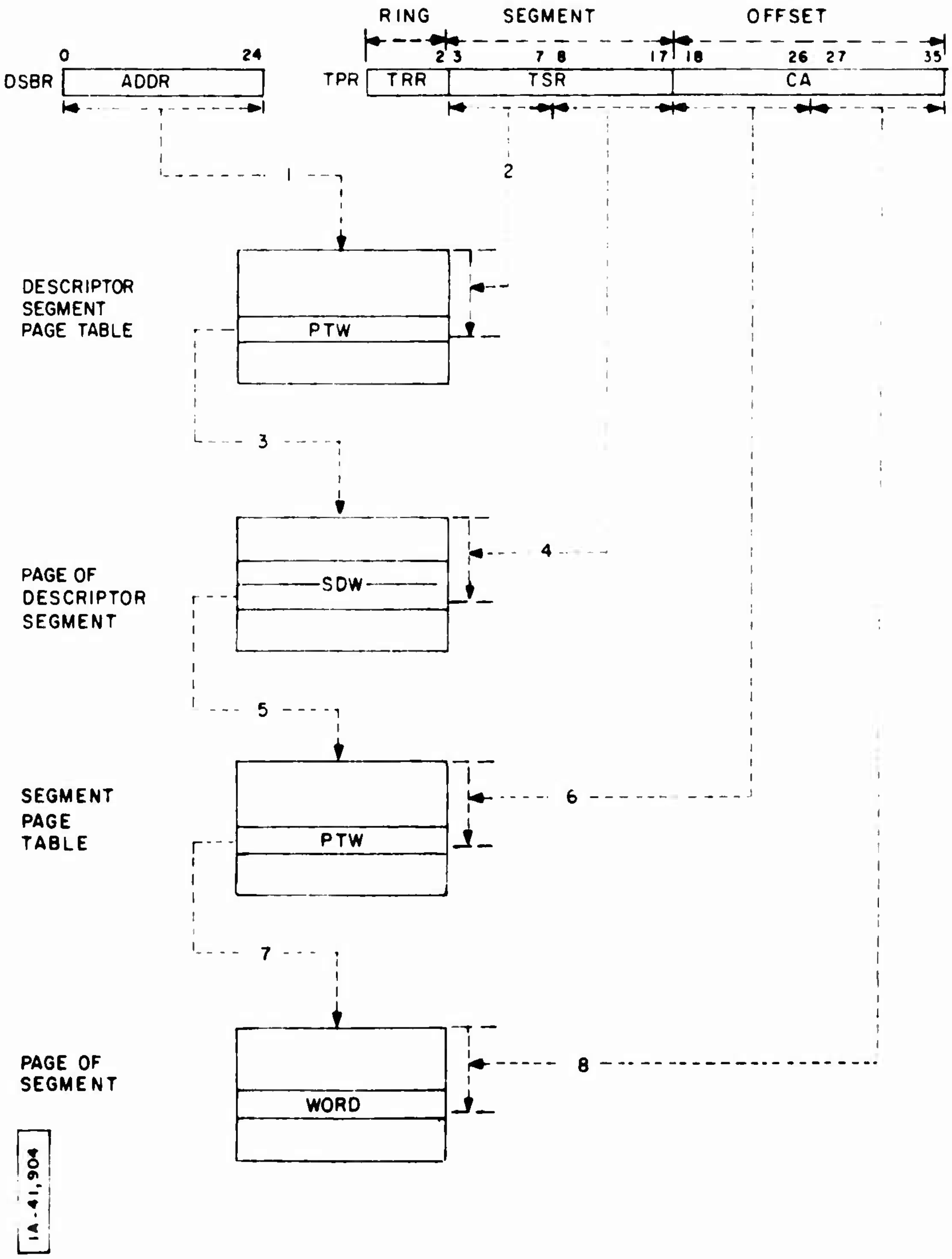


Figure 18. DAT Algorithm

7. This step corresponds to step 3, where the actual address of the page containing the word referenced is determined.

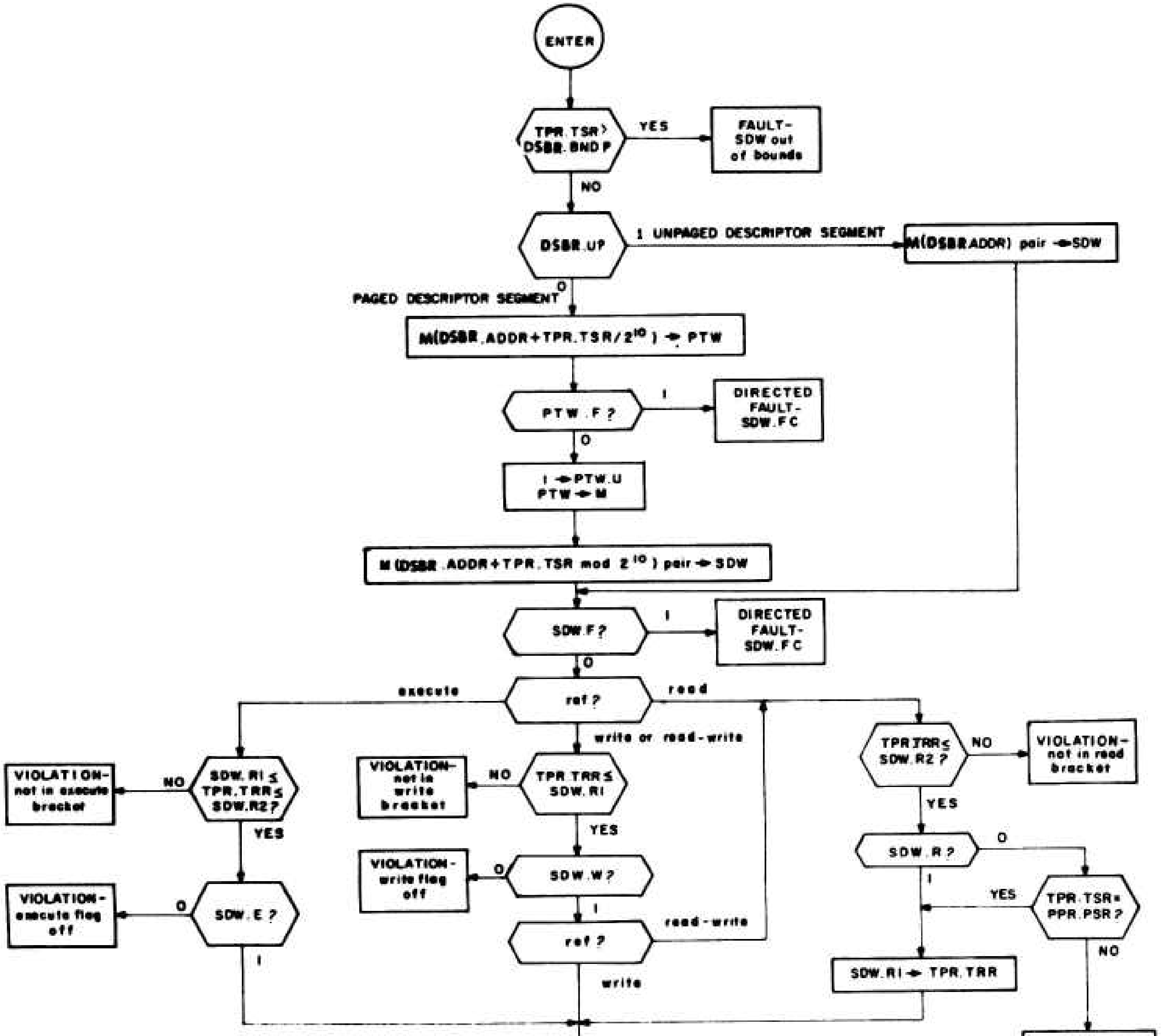
8. The low 10 bits of CA point to the location within the page that contains the word referenced, and the absolute address of this word is thus determined. This absolute address is the output from DAT.

In the worst case three memory references will be needed to determine the address of a word in core. It is possible for the descriptor segment or the segment referenced not to be paged, so that the DSBR or SDW point to the segment itself rather than a page table. In such a case the offset into the descriptor segment or referenced segment is the entire TSR or CA field.

In order to reduce the number of memory references required, a 32 word (60 bit?) associative memory (AM) is used. Sixteen words are used to save PTWs and 16 words are used for SDWs. Each time a PTW or SDW is fetched from core by DAT, it is inserted into the AM, replacing the least recently used word. If a particular PTW or SDW is desired (as identified by its page number and/or segment number which are also stored in the AM), the AM is first searched, and if the word is found the reference to memory need not be made for that word. Since both the segment number and page number can be determined from the virtual address in TPR (steps 2 and 6 above) the associative searches for both the SDW and PTW can be made simultaneously. If the SDW is found, but not the PTW, then DAT must make a memory reference to get the PTW. If the PTW is found, the SDW is not needed, since the PTW contains the address of the page. Along with each PTW in the AM are stored the access bits and ring numbers for the segment of which that PTW is part. Thus access rights may be checked without looking at the SDW. Searching of the AM does not appear to consume any additional time, so that there is nothing wasted if the search fails.

It should be noted that the AM containing PTWs may contain two PTWs from different segments that are from the same relative locations within their segments. This means that PTWs in the AM must be identified not only by their offset in their segment's page table, but by the segment number as well. This then makes it completely unnecessary to reference the SDW if the PTW is found. The actual details of the AM search are not known, nor is the exact format of the AM words. This information is available for

CALLING SEQUENCE: DAT (ref) ⟶ address
ref = read, write, read-write, execute



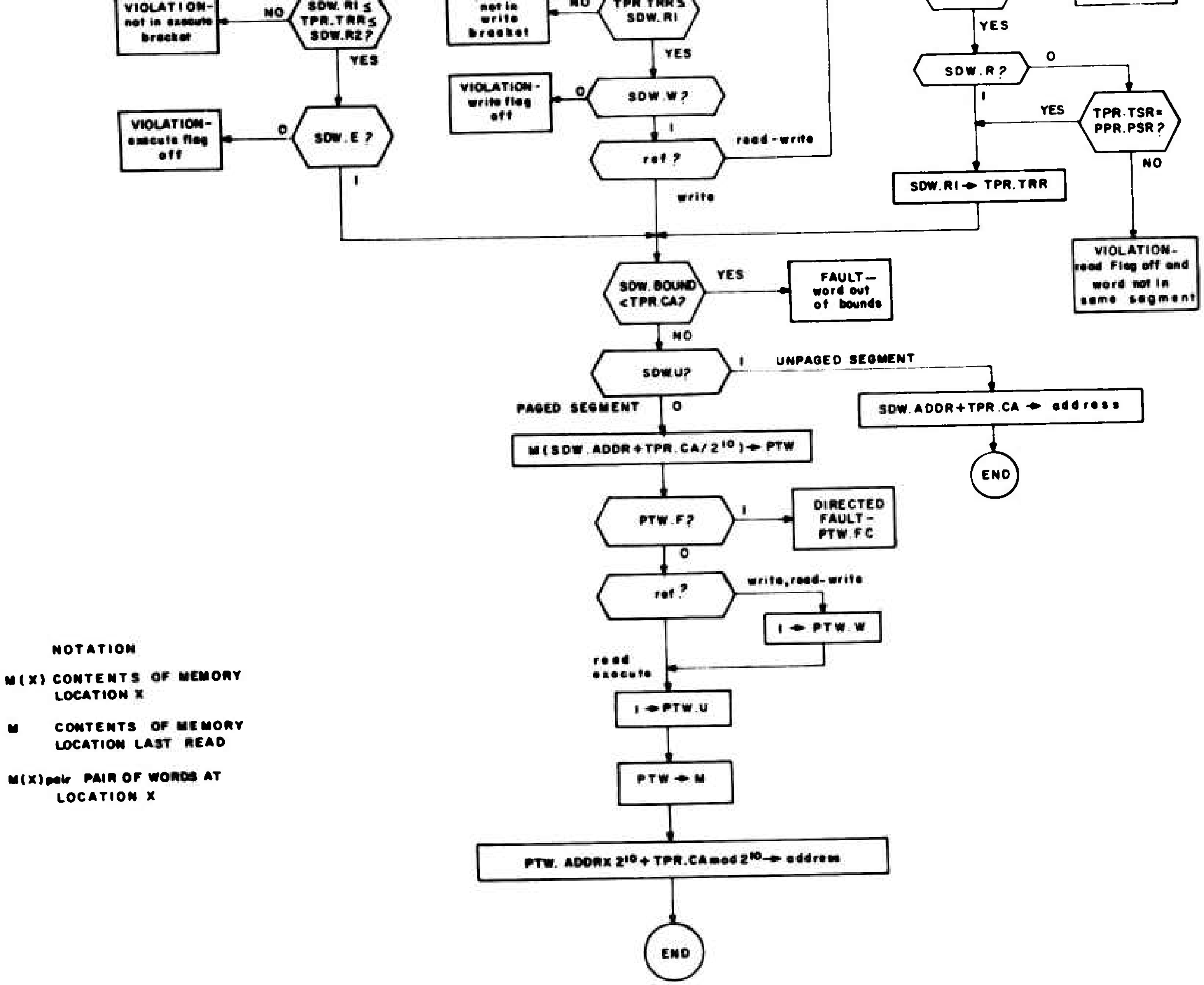


Figure 19. DAT Flowchart

ID-41,913

the 645, and is likely to be similar for the 6180, the basic differences being that for the 6180 only one search of the AM is ever necessary, and that there are more fields (ring access) that must be stored with each word.

Software has the ability to clear the AM, save its contentr, and reload it. Clearing of the AM is necessary whenever a processor removes a page from core. Loading and storing seem only for use in diagnostics.

Figure 19 is a flowchart of DAT, ignoring the use of associative memory. It is assumed that references are all made to core. Reference to a segment with a CALL or RETURN instruction probably should be included in DAT as another type of access because the SDW must be fetched and examined in a manner similar to that for the other types of access. Note that a write into memory is required to set the "written" or "used" bits in the PTW. Such a write must occur even if the PTW is obtained from the AM and the bit was not previously set.

Address Preparation Flowchart. The flowchart in Figure 20 is a partial reconstruction of the effective address preparation and instruction fetch cycles. Some of the less important IT modifications are not included. The flowchart uses certain flags and registers internally (PZ, CT, TRZ, PT). The reference to INS is to a register containing the current instruction and later various temporary tags. IND is a double word register containing the even and odd indirect words. Names of the various parts of IND, such as IND.BITNO or IND.TALLY refer to the corresponding fields in the format of the type of indirect word referenced. Reference to the even or odd IND word is implied by the name of the field or whether the indirect word is an even or odd location.

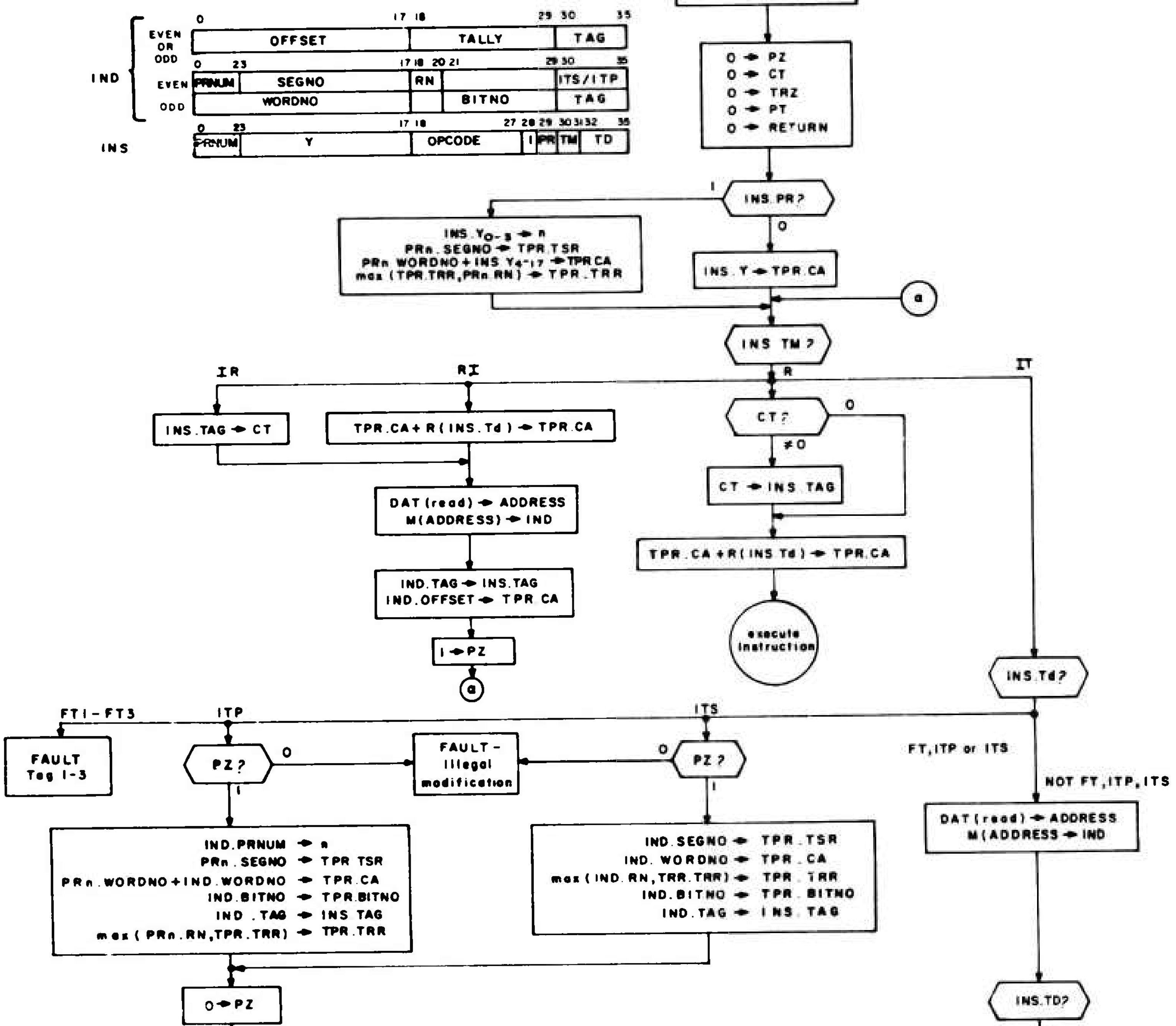
Ideally this flowchart should represent the address preparation exactly. However, the complexity of these sequences forced us to adopt certain simplifications in order to make this task more reasonable. Those points not covered in the flowchart are discussed below.

> Almost every 6180 instruction has certain modifications that are illegal. On the 645, an illegal modification causes an unpredictable result, so such cases were ignored (and would likewise produce inconsistent results from the flowchart). The flowchart was made before it was known that the 6180 generates a fault in every case an illegal modification is used. This feature of the 6180 required that in order to detect such a fault, checks have to be made at various points

NOTATION

R (X) CONTENTS OF REGISTER SPECIFIED BY X

M (X) CONTENTS OF MEMORY LOCATION X



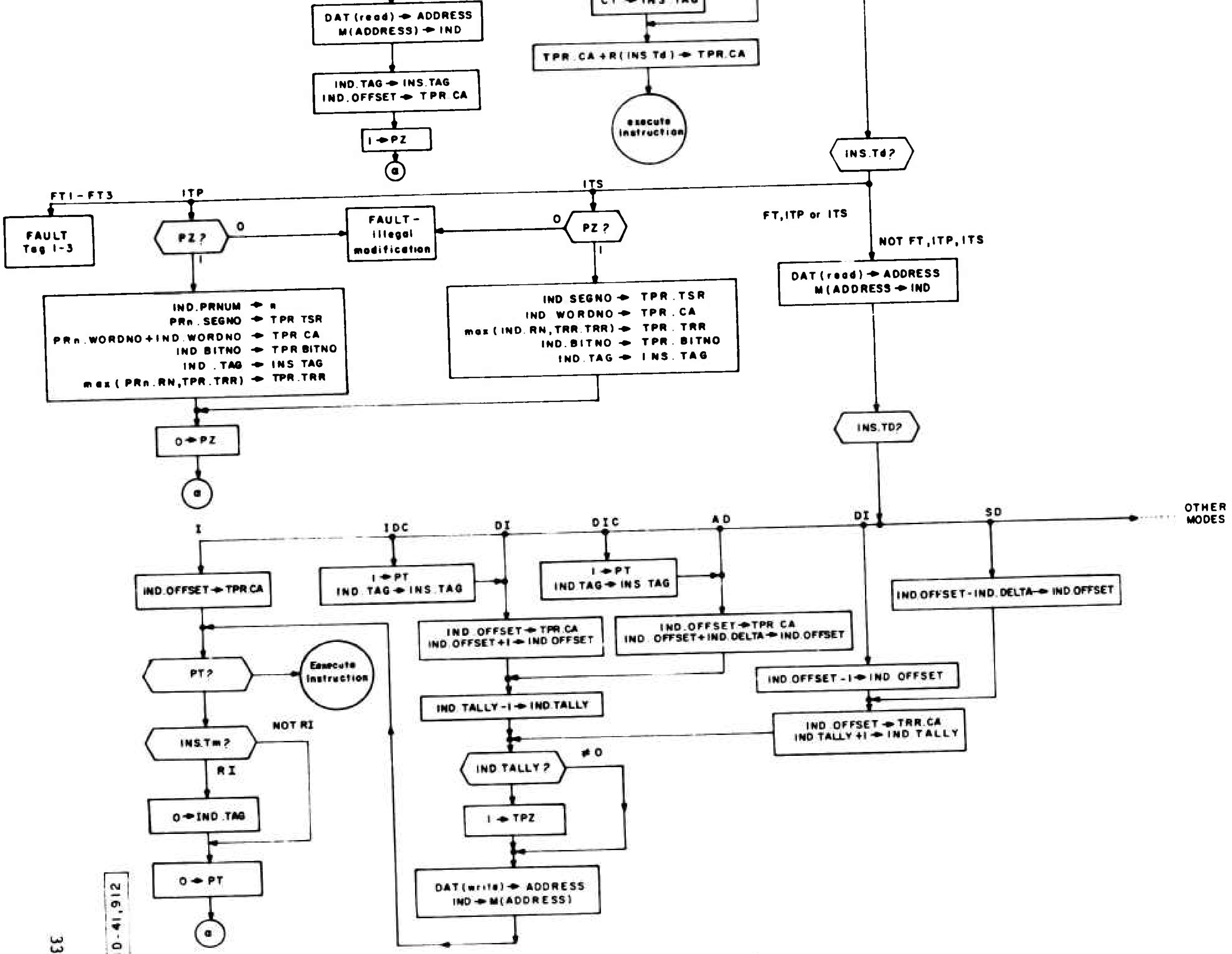


Figure 20. Address Preparation

within the flowchart against the opcode of the instruction. The instructions could be grouped into classes (but not by particular bits in the opcode) for determination of illegal fault-generating modifications.

The actual machine cycle, as discussed previously and illustrated in Figure 7, shows the address preparation split up into independent units. Presumably this split would require many more flags to indicate what the current cycle is. The 645 had five cycle flags, two of which correspond to the PZ and PT flags in the flowchart. The other three were not needed in the flowchart representation. Other flags and registers are also necessary for communication of state between cycles.

The fact that the 645 and 6180 both fetch double words requires additional flags and tests to determine which word is being processed. This requirement not only applies to the even or odd instruction, but also to the even or odd indirect word.

The 6180 and 645 have various repeat modes. A repeat instruction fetches the next one or two instructions and repeatedly executes them until a termination condition specified in the repeat instruction is satisfied. The instructions executed during the repeat mode have many more restrictions placed on the types of address modifications they may use. In addition, address modification is different for each of the two instructions in the repeat double mode, different on subsequent repeats, and a function of bits in the repeat instruction that change address modification for each of the repeated instructions separately. Tests for end of repeat (termination condition satisfied) also have to be made. This confusion required us to temporarily abandon the treatment of address modification during the repeat modes.

In reality, the flowcharts for the address preparation should be an order of magnitude more complex -- not to take care of the failures mentioned above, but due to restrictions in certain modes and slight differences in operation among similar modes. As an example, consider the register modifications DU and DL. Figure 20 doesn't treat these in any special way, but clearly the notation R(INS.Td) makes no sense, and must be handled differently, for this modification. In addition, whenever DU and DL are used, a flag must be set to indicate that no memory fetch is required for the operand.

Preceding page blank

Instruction Cycle and Interrupts -- Detail

As mentioned earlier, the instruction cycle illustrated in Figure 7 is not quite accurate. There are instructions which make reference to DAT as part of the execution cycle (or before execution but after address preparation). Moreover, address preparation is only defined as the construction of an effective virtual address of the operand. For most instructions the operand's absolute address must be determined (requiring a call to DAT) so that the operand can be loaded or stored. Note that the only calls made to DAT during address preparation are for fetching of indirect words. We have attempted to construct a flowchart of what happens after address preparation (Figure 20) and before instruction execution in a form similar to that for the address preparation. This means that the possibility of faults or interrupts being generated is ignored.

Instruction Cycle. Figure 21 is our best representation of the instruction cycle after address preparation that can be made without more 6180 documentation. For purposes of this representation all instructions can be divided into seven groups, each of which is discussed below. Most of this information has either been deduced or taken from Schroeder and Saltzer [1] -- little came from Honeywell documentation. There are several problems which currently prevent us from going into more detail.

1. Instruction does not reference operand.

    This group includes instructions that neither store nor load data into memory nor cause a transfer of control. In this group would be the shift instructions, the repeat instructions, and a few others that only reference registers. There is one inconsistency which should be mentioned at this point. Certain of these instructions (e.g., the shifts) go through the normal address preparation phase, generating page faults or whatever is necessary to determine the effective address of the operand. In the case of shifts, the resultant offset part of the effective virtual address is interpreted as a shift amount, and the operand itself and segment number are ignored. Other instructions, however, such as the repeats, use their tag and y fields for controls, and therefore are not subject to address preparation. This means that a check must be made for instruction type before address preparation so that those cycles are bypassed. Figure 20, on the other hand, assumes that all instructions are treated in the same way. There

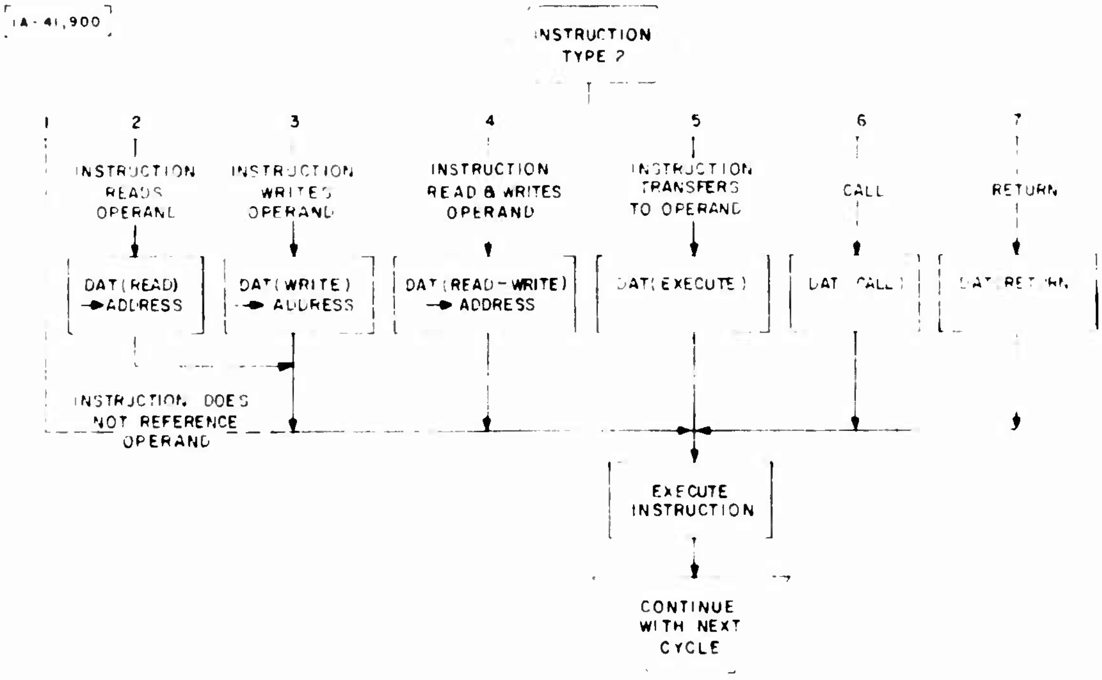

IA-41,900
INSTRUCTION TYPE ?
1
2
3
4
5
6
7
INSTRUCTION READS OPERAND
INSTRUCTION WRITES OPERAND
INSTRUCTION READ & WRITES OPERAND
INSTRUCTION TRANSFERS TO OPERAND
CALL
RETURN
DAT(READ) → ADDRESS
DAT(WRITE) → ADDRESS
DAT(READ-WRITE) → ADDRESS
DAT(EXECUTE)
DAT(CALL)
DAT(RETURN)
INSTRUCTION DOES NOT REFERENCE OPERAND
EXECUTE INSTRUCTION
CONTINUE WITH NEXT CYCLE

Figure 21. Instruction Cycle

are even other instructions that use their tag field for control, but their y field contains the address of the operand. This means that some of the address preparation (PR modification and offset calculation) takes place, but not all of it. Tests for these instructions would have to be made in appropriate places within the flowchart in Figure 20.

2. Instruction only reads operand.

These include the obvious group of loads, arithmetic, and logical instructions that leave their result in a register. For these instructions DAT(read) is called, requesting validation and formation of the physical address of the operand. The actual reading of memory is delayed until the execute instruction box because not all instructions read their operands in exactly the same way. For example, certain load register instructions read 8 or 16 consecutive words. It also seems cleaner to keep the memory reference inside the execute instruction box so that each instruction can do what it wants with the absolute address supplied to it.

3. Instruction writes operand.

   This group contains all the store type instructions that need not read the location to be stored.

4. Instruction reads and writes operand.

   This group contains instructions such as "add storage to accumulator and place result in storage" that must read and write their operands, and thus both read and write access is required.

5. Instruction transfers to operand.

   Treatment of these instructions here is not entirely satisfactory. Figure 21 shows a call to DAT, with check for execute privilege, before "execution" of the transfer. This is not really necessary because execute privilege is already checked for before the fetch of the next instruction (see Figure 20). However, as Schroeder & Saltzer[1] point out, it is desirable to check for legal access before making the transfer, so that a possibly faulting transfer instruction can be located using the old instruction counter. The placement of the call to DAT in Figure 16 allows for this check, but is not good because, in the case of a conditional transfer whose condition is not satisfied, it would be a waste of time to call DAT with accompanying page faults that may occur. This problem could easily be gotten around by treating transfer instructions differently from other instructions, but Schroeder & Saltzer's diagrams seem to indicate that this distinction is not made.

6. & 7. The CALL and RETURN instructions are the only instructions that can change the ring of execution. For the sake of parallelism here it is assumed that the DAT routine has the capability of checking for call or return privilege. It would be possible to perform this function in the "execute instruction" box for CALL and RETURN, but since most of this function (fetching SDW's, checking access) is already

performed by DAT, it seems proper for DAT to do the call and return checks. DAT also changes the current ring number for the new ring of execution. The execution part of CALL and RETURN involves setting certain pointer registers and determining of the stack segment number.

One additional point that should be made is that for transfer, CALL, and RETURN instructions, the absolute address of the operand returned by DAT is not used. Instead the actual transfer consists of loading the virtual instruction counter (PPR) with a new virtual effective address (from TPR) and not with an absolute address.

Interrupts and Faults. The box at the bottom of Figure 7 called "process fault or interrupt" is, after closer examination, much more complex than one would expect. The 645 manual [3] would have one believe that the hardware's processing of an interrupt merely involves taking a snapshot of the CPU state and forcing an XED (execute) instruction into the instruction register tc cause two instructions in certain memory locations to be executed. After putting the XED opcode in the instruction register, the cycle flags need merely be set to force the machine to execute the XED instruction. If neither of the two XED'ed instructions cause a branch, the next instruction to be executed will be the next instruction in the interrupted program.

In reality, the processor never really "lets go" after forcing the XED. The hardware almost expects an SCU (store control unit) in the even location and a transfer in the odd location of the XED'ed pair. If this is not the case the control unit will not be saved and the interrupted program can not be restarted. The handling of these two instructions after an interrupt is different from their normal handling. In fact, if the first instruction is not an SCU and the second not a transfer, the interrupted process is likely to be in trouble. The processor signals a "trouble" fault when something goes wrong with one of these two instructions (such as when the page referenced by the SCU is not in core). These and other conditions mean that special tests have to be made by the hardware during and after address preparation for the SCU and transfer instructions on an interrupt.

SECTION III
EMULATION METHODOLOGY

INTRODUCTION

In order to meaningfully compare the effectiveness of emulations performed on different machines it is necessary for all these emulations to be based on a common methodology. In this section we propose such a methodology. The ultimate goal of this report is to determine the feasibility of emulating the 6180 -- not to present the actual emulation. Therefore this section only discusses those features of the 6180 that may have a significant effect on a "benchmark emulation" such as the one discussed in Section IV. Certain important features of the 6180 are not discussed because their effect on a benchmark is minimal.

An attempt has been made to make this section applicable to all potential microprogrammed emulators, and thus the architecture of the emulator is ignored. However, the current state of the art of microprogrammable processors must be taken into account in order for a methodology to be useful. For this reason we discuss typical hardware limitations that might be encountered, and we consider various ways to deal with these limitations.

GENERAL CONFIGURATION

The first things to determine for any emulation are the level -- machine language or higher level language -- at which the emulation is to take place, and which functional units of the target machine are to be considered. For example, emulation of a "Multics system" at the user level need only duplicate the features available at the terminal. Emulation of a large machine with an I/O channel may include emulation of the I/O channel as well, or may require the I/O channel as a piece of hardware.

The emulation we are considering is an exact machine language emulation. The emulator must be completely compatible with machine language software for the 6180. The only difference allowed is a time difference: we are assuming that 6180 software is not time dependent enough for a slowdown to affect its proper execution. Certain common practices of IBM 7090 days, such as software timing loops, are assumed not to occur. It may be that for proper operation of some external devices, two instructions must be executed within a specified time of each other, but this only requires that the emulator be "fast enough" to handle those devices by normal programming methods. The slowdown factor that might be considered reasonable or maximum for the emulator cannot at this point be determined. There are too many variables that can affect ultimate performance of the 6180 in its normal Multics environment, (memory size, user load, types of applications, design of system software) and the emulator has no control over most of these. Whenever we have to make a decision between apparent cost or complexity and speed, we will rely more on intuition than calculated performance figures.

The 6180 consists of several functional units as discussed in Section II. Since the separation of these units is not detectable by software, there is no reason for dividing the emulator into such units. The possibility of asynchronous operation of the functional units is not important since such operation only affects speed. We will therefore assume that one microprogrammable CPU, or one "emulator," is sufficient to emulate one processor. The usual 6180 configuration of several processors and main memory is not software independent (software is aware of the number of processors) so a separate emulator would be required for each processor. However, the ultimate purpose in having multiple processors is only for increasing performance which is again not very important at this time. I/O channel operation must eventually be considered because I/O channels are necessary and the I/O multiplexors are treated by software as separate asynchronous units. Emulation of these units is not discussed in this report due to its minimal effect on a benchmark. At some future time the tradeoff between I/O multiplexor emulation and I/O hardware has to be determined.

Although it is not necessary to separate the emulator into functional units, one of the units - the appending unit - performs such a complex function in essentially "zero" time that a microprogrammed emulation of it would not be feasible. The associative memory is the main reason: without it, three memory references are required just to determine the address of a location in virtual memory. A slowdown by a factor of three or four may not be unsatisfactory, but when one considers that this slowdown will be added to any other slowdowns realized in the emulator, associative memory hardware becomes a necessity. Moreover, the structure of dynamic address translation is such that there is not much the processor can do while waiting for a PTW or SDW to be read from memory. It seems fairly certain that any emulation will require DAT hardware that duplicates the function of the DAT subroutine illustrated in Figure 19.

Another difficulty with the 6180 is the large number of registers available to the programmer and required by hardware. If the machine doing the emulation has sufficient register capacity, this presents no problem. However, it is more likely that some or all of the registers in the 6180 will have to be put in places such as core, control store, or other memory units. Judicious assignment of registers will have an important effect on emulator performance.

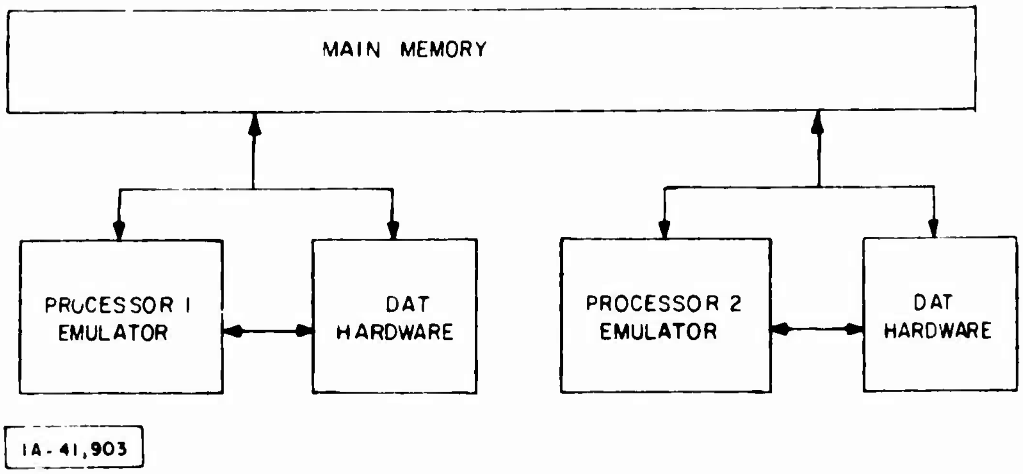


Figure 22. Emulator Configuration

The complete emulator configuration, assuming DAT hardware is required, is shown in Figure 22. A dual processor is used as an example. For any one particular source machine, the processor emulator might require additional hardware units. Note that the DAT hardware is complicated by the fact that it too must read and write memory (for fetching PTWs and SDWs and writing PTWs). However, DAT does not really have a separate port into memory because it never makes a reference at the same time as its processor.

Besides the DAT box, additional hardware is needed for implementing the calendar clock and timer registers. It can be safely assumed that a typical off-the-shelf microprogrammable machine does not have such registers, and that emulation of these timers through microprogramming or 1-microsecond interrupts is too time consuming.

All other configuration features of the 6180 (e.g., interleaving, overlap) that normally just help to speed up performance are not considered at this point in the emulation, but might be ultimately required in a given simulation. Note that software can detect and use interleaving.

MICROPROGRAM ORGANIZATION

In Section II and Figures 6 and 7, it was noted that the instruction cycle of the 6180 is not like that of a "classical" machine because faults and interrupts are allowed during the address preparation and not just between instructions. If the structure were as in Figure 6, microprogramming the address preparation would be straightforward, though complex. There would be no need to find out exactly how the 6180 prepares its addresses as long as correct results were obtained. Programming of the instruction execution would also be straightforward.

The fact that the address preparation is split into several defined cycles (of which there are 5 on the 645) puts a serious restraint on the freedom allowed to the microprogrammer. He must now organize his microprogram so that faults and interrupts occur at many specific points in the instruction cycle. In addition, he must make sure that a return from an interrupt or fault will restore the processor to the point it left off. The worst restraint seems to be involved with the control unit status. At an interrupt or fault, eight words of packed bits of information must be available for software to save with the SCU instruction. In programming the address preparation of Figure 20, it may not be very convenient for the microprogrammer to represent his microprogram state in exactly the same way done by the 6180 hardware, but it is absolutely necessary that the state stored by the SCU instruction look the same in core. One can imagine spending a great deal of time packing flags and bits into the 8 words at each interrupt so that they appear in the proper format for software. Alternatively, one can imagine firmware continuously keeping its own state in the right format ready for storing, but comsuming time on each instruction interrogating its flags because they are not conveniently placed in its own internal storage elements. The trick is to find the proper balance between representation of microprogram state in a defined format and packing and unpacking bits at interrupts -- otherwise performance is likely to suffer.

The flowchart of the address preparation in Figure 20 must be reorganized into "cycles" as in Figure 7. In addition, the end of the address preparation, represented in Figure 21, has to be worked in so that all are consistent. The cycle flags, shown in Figure 3

(PI, PA, etc.) must be used to determine which of the address preparation cycles is currently being performed. Note that Figure 20 does use several of the flags and temporary registers for its own state storage, but only to determine when certain conditions are illegal and to define the path of flow under certain conditions. Their use in the flowchart probably corresponds to their use in the 6180, except that the flowchart does not save or restore their values on faults. For a full explanation of these flags and registers, refer to the 645 manual [3].

In Figure 7, there is one call to DAT within each cycle. Presumably there is no reason to define a cycle as separate unless it contains a call to DAT. It was noted previously that, although the fault is generated during DAT, it is not detected until the end of the cycle. In other words, the end of cycle is forced when any fault occurs in DAT. At time of detection, the interrupt handler is invoked. Later, upon restoration of the control unit status the process should resume at some appropriate point within the cycle. Ideally the return of control should be to the DAT algorithm that caused the fault, with the same arguments (TPR and type of access) that it had previously. (Assuming, of course, that the TPR and other control unit bits have not been altered by software.)

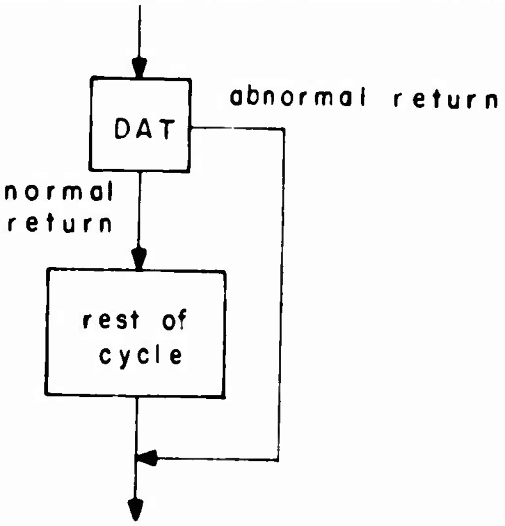


Figure 23. Typical Machine Cycle

What is required is an accurate portrayal of a typical cycle, with the location of the DAT call specifically indicated. The cycle could be thought of as a call to DAT first, followed by the rest of the cycle, assuming that the TPR and access request are saved in the control unit's status. In the 645 the SCU does save the TPR, but not the access request, though it is probable that the type of access required can be deduced from the cycle designation. With this scheme each cycle can be drawn as in Figure 23. The DAT call has a normal return, which allows continuation of the cycle, and it has an abnormal return, in the case of a fault, that goes to the end of the cycle.

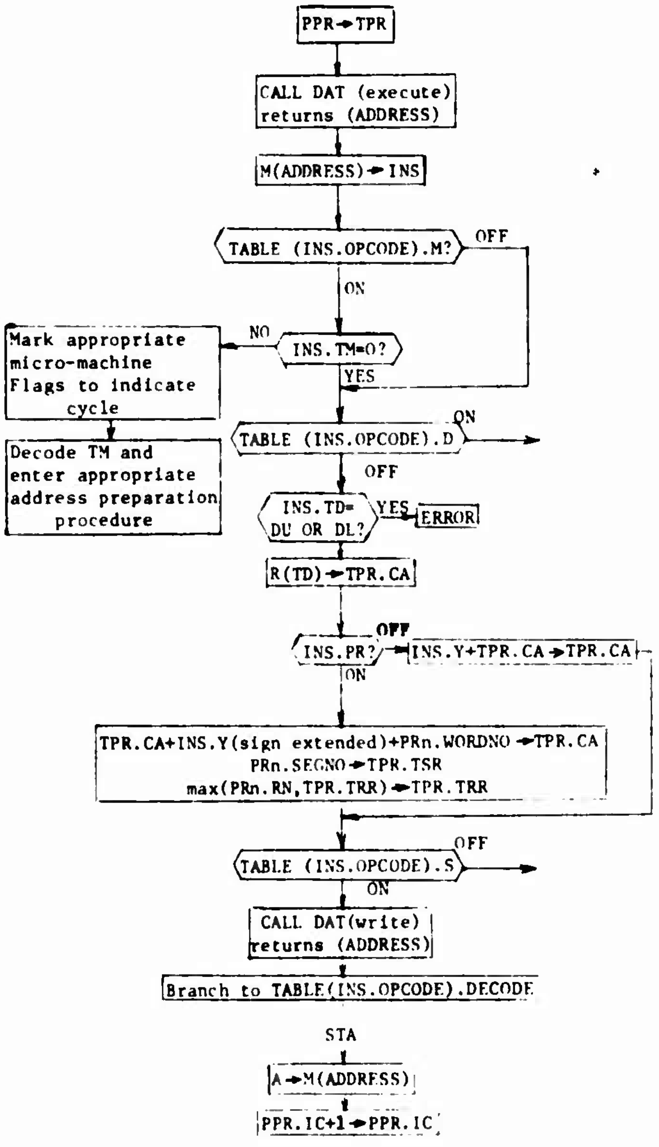
SECTION IV

BENCHMARK EMULATIONS

INTRODUCTION

Given the precise description of the 6180 and a method for the emulation of that machine, it is now possible to discuss the emulation of the 6180 by the three selected microprogrammable processors. In a benchmark emulation of the type to be developed here, it is not necessary, and indeed it would be highly impractical, to present the detailed emulation. The goal is to emulate a representative machine function to assess the feasibility of the emulation and to gain a first-order approximation of the efficiency of such an emulation. The measure of efficiency used here will be expressed as the time to emulate the representative function relative to the time the 6180 takes to perform the same function.

An examination of machine language code generated for PL/I programs in Multics indicates that almost every reference to a data location is through a pointer register (PR). For this reason, and because a store is somewhat more involved than a load, the function chosen as representative is the store A instruction with PR modification and no indirection. Simple register modification is assumed. The instruction fetch and execute cycle for this example are shown in Figure 24. The TABLE and INS data bases used in this flowchart are described in TABLE II as PL/I-type structures. One main simplification introduced is that the call to the DAT algorithm produces no faults. In making the comparison, it is assumed that indirection through the pointer adds no time to the instruction execution cycle on the 6180. Thus the benchmark emulations must follow the same path through the instruction cycle as that taken by the STA on the 6080 to allow a comparison with the 6180.


PPR→TPR
CALL DAT (execute) returns (ADDRESS)
M(ADDRESS)→INS
TABLE (INS.OPCODE).M?
OFF
ON
Mark appropriate micro-machine Flags to indicate cycle
NO
INS.TM=0?
YES
Decode TM and enter appropriate address preparation procedure
TABLE (INS.OPCODE).D
ON
OFF
INS.TD= DU OR DL?
YES
ERROR
R(TD)→TPR.CA
OFF
INS.PR?
INS.Y+TPR.CA→TPR.CA
ON
TPR.CA+INS.Y(sign extended)+PRn.WORDNO→TPR.CA
PRn.SEGNO→TPR.TSR
max(PRn.RN,TPR.TRR)→TPR.TRR
OFF
TABLE (INS.OPCODE).S
ON
CALL DAT(write) returns (ADDRESS)
Branch to TABLE(INS.OPCODE).DECODE
STA
A→M(ADDRESS)
PPR.IC+1→PPR.IC

Instruction counter to temporary pointer register.

Call DAT for execute access. TPR is implied argument to DAT. ADDRESS of instruction is returned.

M is memory location, INS is instruction.

Check M flag to see if address modification is allowed on this instruction.

Check to see if indirection is specified. If so, then further address preparation cycles are needed.

If D flag is on, DU or DL (immediate) modifications are allowed, in which case this can't be a store instruction.

Otherwise DU or DL are illegal.

Get contents of register specified in TD field

PR modification, indirection through a pointer register, is specified in bit 29. Otherwise, INS.Y has offset.

If PR is specified, get WORDNO or PR added to 15 bits of INS.Y to compute offset. SEGNO of PR has new segment number and RN can specify a greater ring number.

If S flag is off, this is not a store type instruction.

If store type, ask for write access to location and get its ADDRESS.

Branch to micro-routine for opcode.

This is a STA instruction.

Store A in memory

Increment instruction counter.

Figure 24. Instruction Fetch and Fxecute Example (Store A)

TABLE II

Data Bases for Instruction Fetch and Execute

```
1  TABLE (Array) FIXED BIN (22),

     2   M    BIT(1),         PERFORM MODIFICATION BIT

     2   X    BIT(1),         NO MODIFICATION ALLOWED

     2   D    BIT(1),         DU,DL MODIFICATION

     2   C    BIT(1),         SC,CI  SCR ALLOWED

     2   E    BIT(1),         EIS INSTRUCTION

     2   S    BIT(1),         STORE TYPE

     2   DECODE   FIXED BIN (16);    POINTER TO MICROCODE


1  INS FIXED BIN (36),

     2   Y    FIXED BIN (18),

     2   OPCODE   FIXED BIN (10),

     2   EIS BIT (1),

     2   PR  BIT (1),

     2   TM  FIXED BIN (2),

     2   TD  FIXED BIN (4);


TPR   |
      } As previously defined (see Figure 3)
PPR   |
```

THE BURROUGHS D-MACHINE

Description of the D-Machine

The Burroughs D-Machine, or Interpreter as it is sometimes called, is a modular, two-level microprogrammable computer with the capability for sophisticated interconnections between individual D-Machines. The machine is best described by Reigel et al [11] and Bingham et al [12] . The switching interlock, used to connect many D-Machines for multiprocessor emulations, is described by Davis et al [13] . The reader should consult the aforementioned references for detailed descriptions of the architecture and operation of the D-Machine. The following description gives an outline of those D-Machine features germane to the benchmark emulation.

The block diagram of an interpreter is shown in Figure 25. The important characteristics of the interpreter are the two-level memory and the restricted number of key data paths and general registers.

The two-level microprogram and nanoprogram memory of the D-Machine provide a wide control word (54 bit nano-memory width) without the disadvantage of needing large amounts of storage for the microprogram. The 16 bit microprogram word is either a pointer to a nanoprogram word or a literal. Thus, any given emulation will be characterized by a small set (500 to 4000) of nanowords and a larger microprogram.

The other important characteristic, that of limited data paths and general registers, works to the detriment of a sophisticated emulation. Where the emulated machine is characterized by a large machine state (as the 6180 is) or where the emulation requires complex microprogram interconnections (like stacked calls to microprogrammed routines), the D-Machine must be augmented to perform the emulation efficiently. These two deficiencies must be partially overcome for the 6180 emulation.

Honeywell 6180 Emulation

To perform the 6180 emulation in an efficient manner, the limitations alluded to above must be overcome. The lack of general registers can be alleviated by the proposed D-Machine enhancement to provide functional elements addressable by D-Machine logic. Such an arrangement is shown in Figure 26 and exists on an operational D-Machine. In the discussion of the emulation, these additional

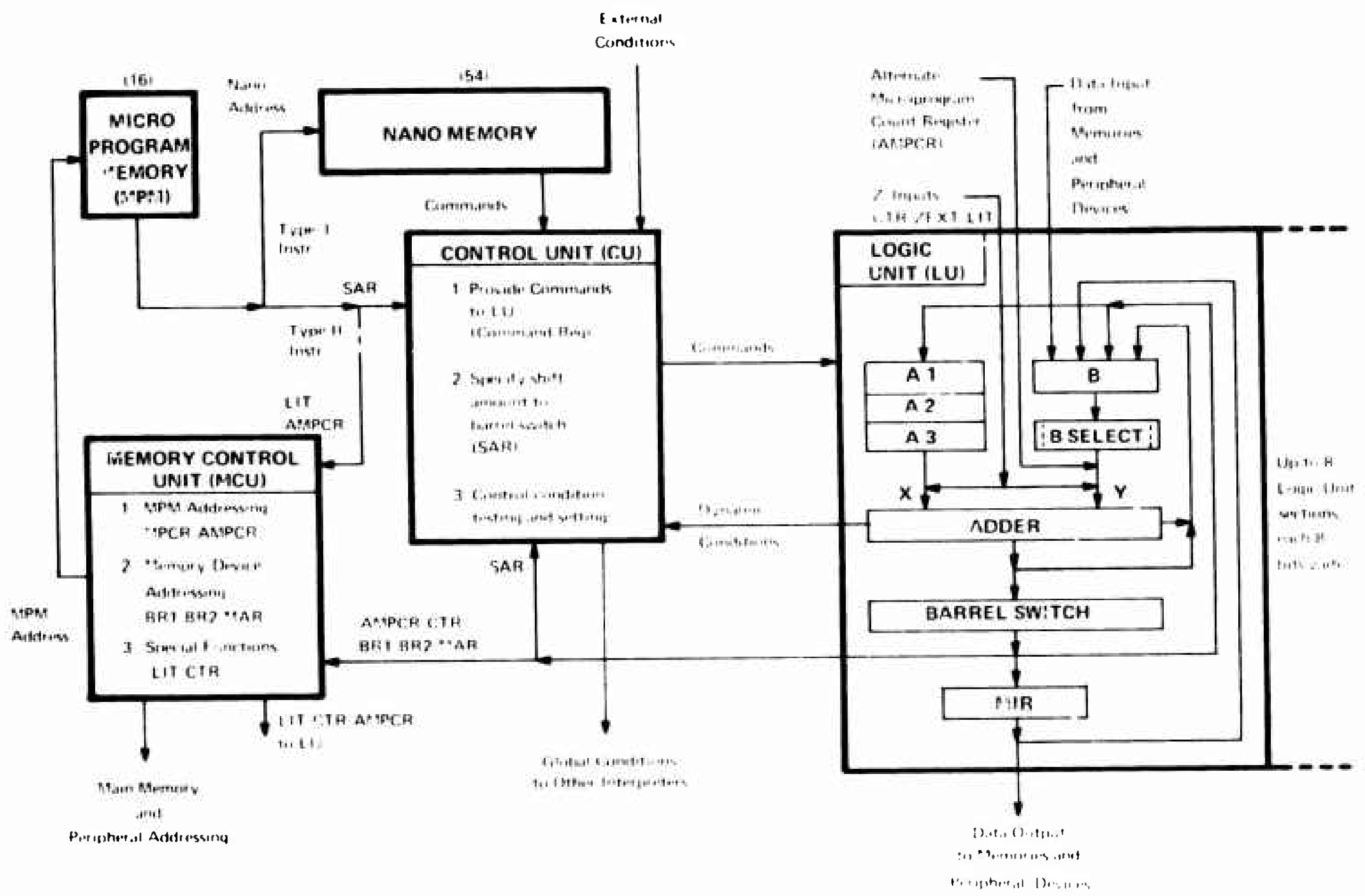

(16)
MICRO PROGRAM MEMORY (MPM)
Nano Address
(54)
NANO MEMORY
External Conditions
Commands
Type I Instr
SAR
Type II Instr
LIT AMPCR
CONTROL UNIT (CU)
1 Provide Commands to LU (Command Reg)
2 Specify shift amount to barrel switch (SAR)
3 Control condition testing and setting
MEMORY CONTROL UNIT (MCU)
1 MPM Addressing MPCR AMPCR
2 Memory Device Addressing BR1 BR2 MAR
3 Special Functions LIT CTR
MPM Address
SAR
AMPCR CTR BR1 BR2 MAR
LIT CTR AMPCR to LU
Main Memory and Peripheral Addressing
Global Conditions to Other Interpreters
Commands
Dynamic Conditions
Alternate Microprogram Count Register (AMPCR)
Z Inputs CTR /FX1 LIT
Data Input from Memories and Peripheral Devices
LOGIC UNIT (LU)
A 1
A 2
A 3
B
B SELECT
X
Y
ADDER
BARREL SWITCH
MIR
Up to 8 Logic Unit sections each 8 bits wide
Data Output to Memories and Peripheral Devices

Figure 25. D-Machine Block Diagram

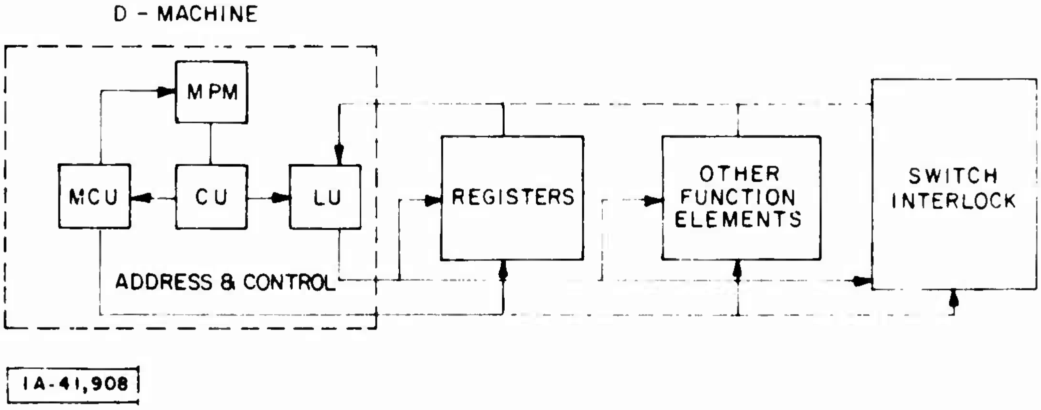


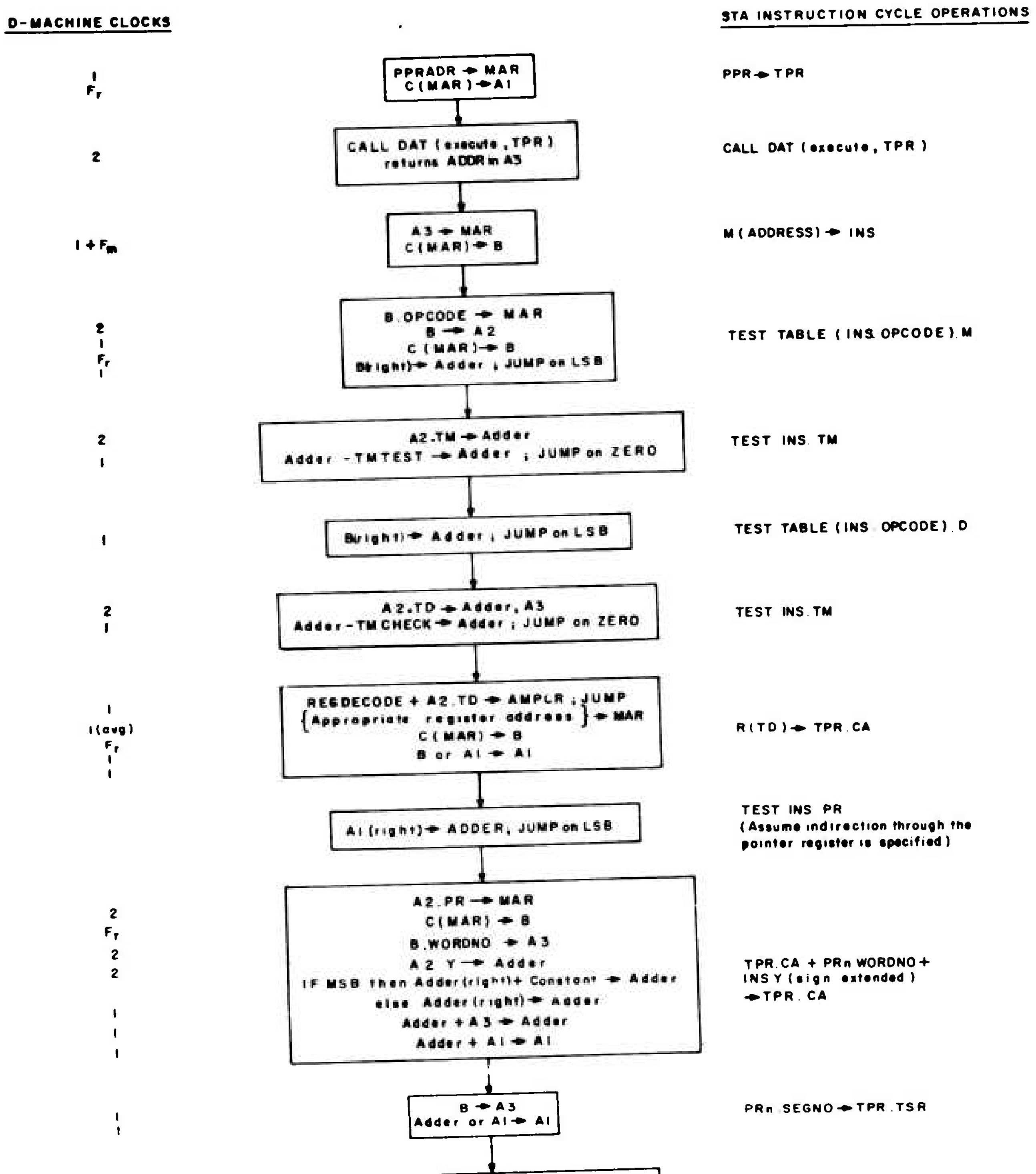
Figure 26. Function Elements Added to D-Machine

registers will be assumed to exist, although no initial assumption will be made about their access times. Also, it is assumed that the TABLE data base is stored as a functional element.

The DAT (Dynamic Address Translation) algorithm must be performed by the D-Machine as well. To simplify the emulation it will be assumed that the necessary functional elements for the DAT emulation, such as associative memories are provided.

With the above assumptions, it remains only to go througħ the algorithm (the STA instruction) of Figure 24, making estimates of the time required by the D-Machine for each activity and commenting on the manner in which the activity is emulated.

The flowchart for the D-Machine emulation of the 6180 STA instruction is shown in Figure 27. The figures on the left of Figure 27 indicate the D-Machine clock times needed for each D-Machine operation. The symbols F_r and F_m stand for the time for a function register fetch and the time for a main memory fetch, respectively. The field designations used on the D-Machine registers refer to the field within the 6180 word currently being emulated in that register. For example, B.OPCODE refers to the INS.OPCODE field when the B register contains the INS data word. From the timings in Figure 27 it can be seen that the STA instruction emulation takes 48 D-Machine clock times. If the F_r time is assumed to be the same as a D-Machine clock time (not an unrealistic assumption),


D-MACHINE CLOCKS
STA INSTRUCTION CYCLE OPERATIONS
1
F_r
PPRADR → MAR
C(MAR) → AI
PPR → TPR
2
CALL DAT (execute, TPR)
returns ADDR in A3
CALL DAT (execute, TPR)
$1 + F_m$
A3 → MAR
C(MAR) → B
M(ADDRESS) → INS
2
1
F_r
1
B.OPCODE → MAR
B → A2
C(MAR) → B
B(right) → Adder ; JUMP on LSB
TEST TABLE (INS.OPCODE).M
2
1
A2.TM → Adder
Adder -TMTEST → Adder ; JUMP on ZERO
TEST INS.TM
1
B(right) → Adder ; JUMP on LSB
TEST TABLE (INS. OPCODE).D
2
1
A2.TD → Adder, A3
Adder-TMCHECK → Adder ; JUMP on ZERO
TEST INS.TM
1
1(avg)
F_r
1
1
REGDECODE + A2.TD → AMPLR ; JUMP
{Appropriate register address} → MAR
C(MAR) → B
B or AI → AI
R(TD) → TPR.CA
AI (right) → ADDER, JUMP on LSB
TEST INS.PR
(Assume indirection through the pointer register is specified)
2
F_r
2
2
1
1
1
A2.PR → MAR
C(MAR) → B
B.WORDNO → A3
A2 Y → Adder
IF MSB then Adder(right) + Constant → Adder
else Adder(right) → Adder
Adder + A3 → Adder
Adder + AI → AI
TPR.CA + PRn.WORDNO + INS.Y (sign extended) → TPR.CA
1
1
B → A3
Adder or AI → AI
PRn.SEGNO → TPR.TSR

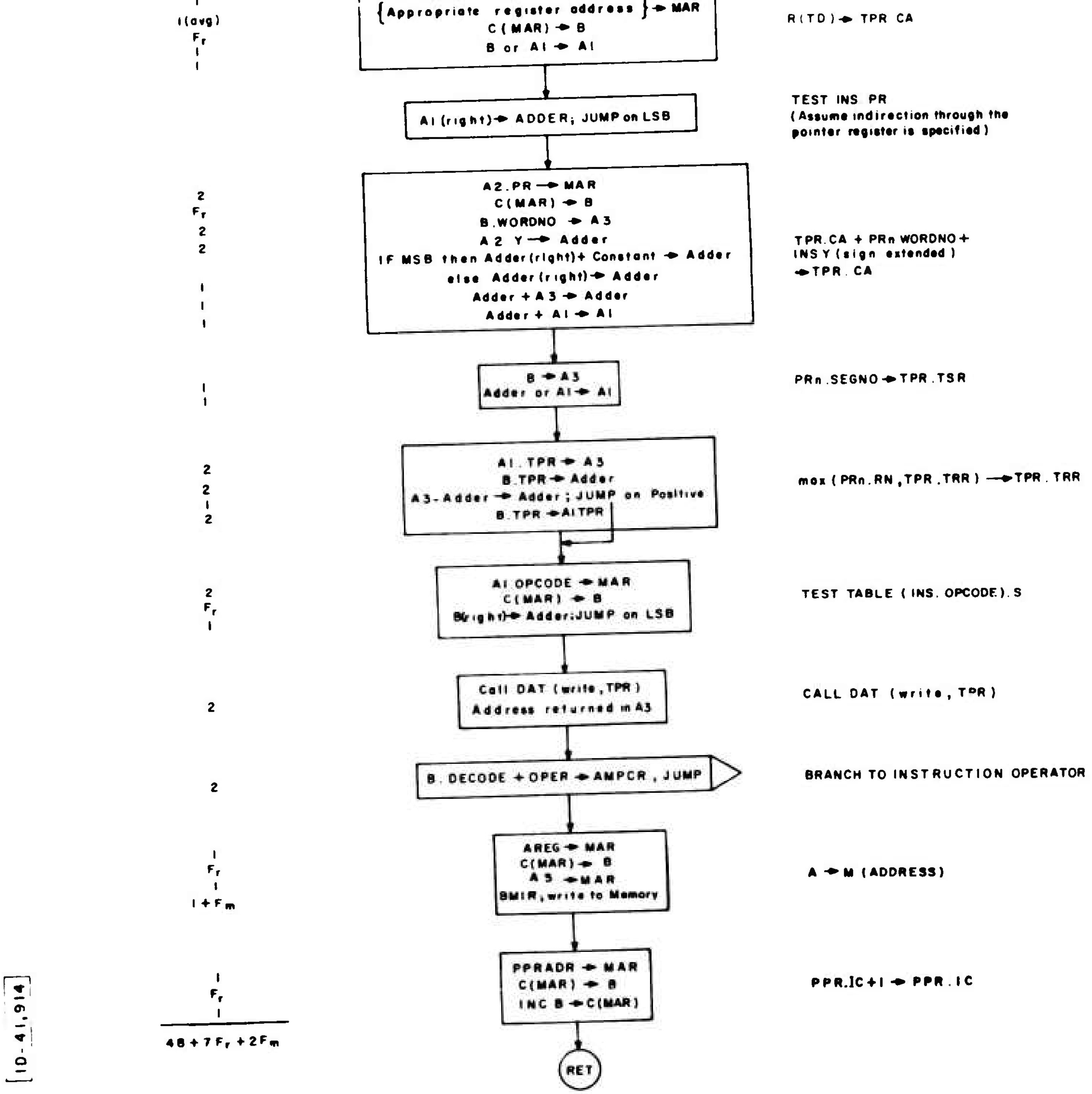


Figure 27. D-Machine Emulation Flowchart

then the STA instruction takes 55 clock times which at a 3 MHz or 5 MHz clock rate represents 18 μsec or 11 μsec respectively. (Remember that the STA instruction takes 1 μsec in a 6180.) By resorting to the ECL circuitry, instead of TTL, and paying 50¢/bit for the nanoprogram memory, D-Machine clock speeds of 15-20 MHz could be achieved resulting in 3.5 μsec-2.7 μsec execution times for the STA instruction emulation. In a discussion with Burroughs personnel, we were told that the transition to the higher speed circuitry is feasible.

THE NANODATA QM-1

Below is a brief description of the features of the QM-1 unique to that machine, and details of a benchmark emulation. For a complete description of the QM-1 and its operation, see the QM-1 Hardware Level User's Manual [10].

Description of the QM-1

The QM-1 is both micro and nanoprogrammable. Nanoprogramming is one level below microprogramming in the sense that microprogramming is a level below machine language programming. In the QM-1, the hardware structure, which represents register units, data paths, and memory units, is predetermined. The control of these units, however, is completely under the control of the nanoinstructions. A nanoinstruction is composed of many nanoprimitives that specify operations to be performed in parallel in a single clock time. A nanoprimitive, for example, may specify that the data available on a particular bus be gated to its destination. Theoretically a maximum amount of parallelism can be achieved because the nanoinstruction is sufficiently large so that all events that can possibly take place simultaneously without conflict can be specified in one nanoinstruction. The basic nanoinstruction is 360 bits wide, and is divided into five 72-bit fields designated K, T1, T2, T3 and T4. All the T fields are identical, and at any one time only the K and one of the T's are active. A T field in execution is referred to as a T-step. The T fields specify gating functions and control and the K field contains such things as constants and function specifications to be used by the T-steps. Since the K field must be shared among all four T-steps, some T-steps often can not make use of the K field required by the other T-steps in that nanoinstruction. Under such conditions a T-step must be wasted.

Preceding page blank

The QM-1 has an 18-bit structure for its main data paths, and a 6-bit structure for auxiliary data paths. The 18-bit structure consists of 32 general purpose local store registers, main memory, control store, shifter, adder, 32 dedicated and general "external" registers, and 14 buses connecting these units. One end of each bus may connect to any one of the local store registers. The other end of each bus connects the local store register, as either source or destination, to another unit. Two of the buses may have their other end attached to any one of the external registers.

The 6-bit structure is used for both control and data. F-store, which consists of 32 6-bit registers, has 14 registers assigned for bus control. For each 18-bit bus, one of these 14 F registers is used to determine to which of the local store registers that bus is attached. Similarly two more F's determine which of the external registers the other ends of two of the buses are attached to. Within a T-step, one can specify which F is to be loaded with a value or constant contained in the K field (thus selecting the local or external store register to which a particular bus is assigned), and which buses are to be gated to their destination. There is also a 6-bit arithmetic unit for 6-bit manipulation. Transfers of data between the 6-bit structure and 18-bit structure are made through local store register 31. This register is broken up into three 6-bit fields, separately addressable in a manner similar to the F's.

A basic T-step executes in about 80 nanoseconds. The results of a gating action specified in a T-step are not usually available in the following T-step, but in the T-step after that. If no use can be made of the extra T-step, a stretch option is available to double the time of the current T-step in order to make data available for the next T-step. This stretch option saves no time, but it does save nanostore for the unused T-step. Certain operations always must be placed in a stretched T-step to work. A nanoprogrammer has to be very careful in defining his register usage and order of operation so that unused T-steps and time periods are minimized.

The designers of the QM-1 intended that control store be used as a general purpose high speed storage for the microprogram and data. For an emulation of a machine with more than a few registers, the register values can be stored in control store. The actual microinstructions used by the QM-1 are, of course, defined by the nanoprogram. However, various features encourage one to adopt one or more of several microinstruction formats. This is not seen to be much of a restriction or disadvantage, since the choices available are the most logical.

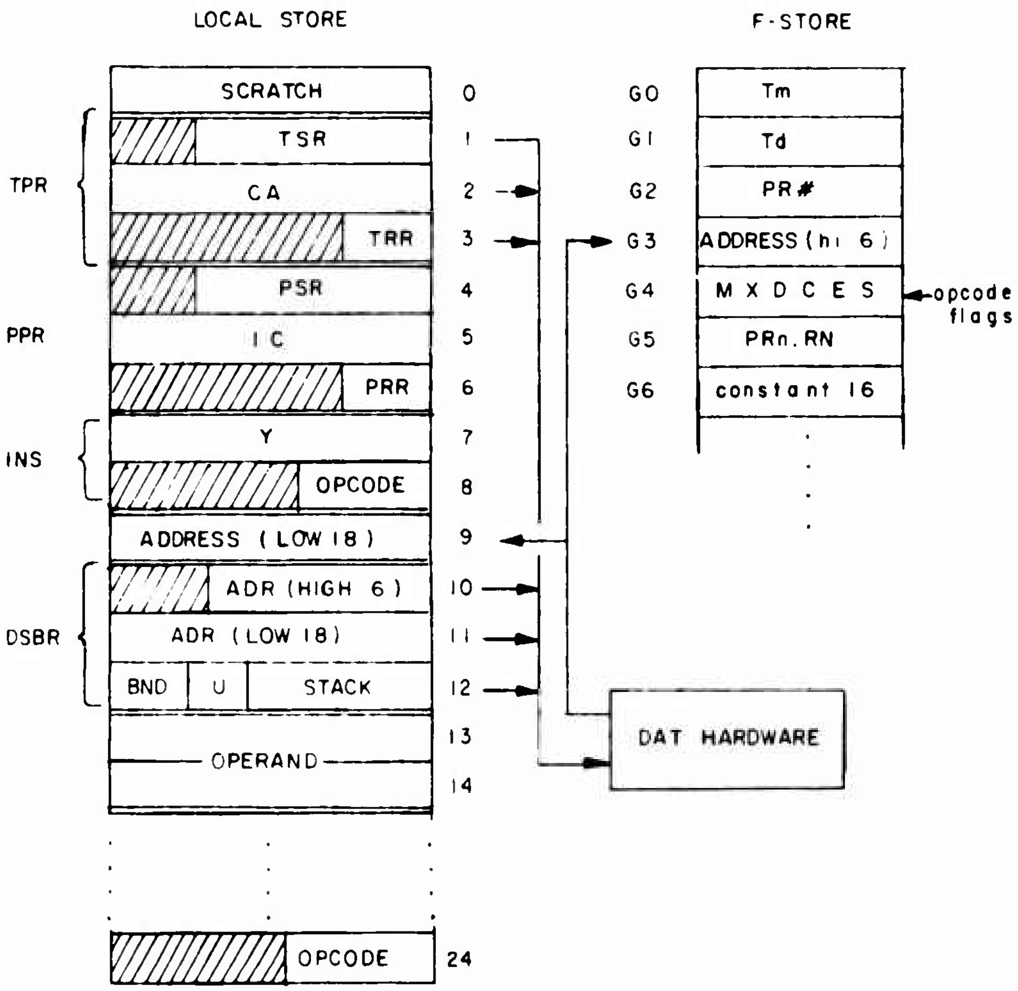
Honeywell 6180 Emulation

The benchmark emulation on the QM-1 cannot be completely defined without extensive experience in nanoprogramming and consultations with Nanodata Corp. The skills that need to be developed for efficient nanoprogramming could not be mastered in the short amount of time available to us for determination of this benchmark. However, feasibility could be determined and several figures have been arrived at.

The basic hardware structure of the QM-1 to be used for a 6180 emulation is almost identical to the standard QM-1, since the machine is not modular and expandable like the D-Machine. Some changes, however, are incorporated as practically necessary to obtain reasonable efficiency. Most important are the expansion of the main memory word size to 36 bits instead of 18, and expansion of addressing capability to 24 bits. The memory unit in the standard QM-1 empties its output data into a buffer unit (referred to as the RMI unit) that performs one of several rotate and mask operations selectable by the nanoinstruction that reads memory. This unit can be expanded to hold 36 bits of memory data, either half of which could be selected as standard 18-bit data. Memory input data could be similarly buffered in two 18-bit halves. The rest of the 18-bit structure of the QM-1 is ideal for emulation of a 36-bit machine. Expansion of addressing range to 24 bits requires assignment of one of the F registers as a 6-bit high order extension of the 18-bit memory address now gated into the memory bus.

This emulation requires specially built DAT hardware for the QM-1. Control of this DAT hardware is accomplished by adding bits to the K or T fields that request access and initiate action. The DAT hardware makes references to the TPR and DSBR stored in defined local store registers, and returns its address in another defined local store register and F-store.

Assignment and layout of control store and F registers pertinent to this benchmark are shown in Figure 28. The last twelve F registers are general purpose and known as G0-G11. The interfaces to the DAT hardware are also shown. G4 is used to hold the 6 bits designating opcode type for purposes of checking illegal address modes and determining whether the instruction is a load or store type.



IA-41,901

Figure 28. Register Layout for QM-1 Emulator

In addition to register storage in the QM-1, the lower locations of control store are used to hold the pointer registers (PRs), index registers, and the A and Q registers. There may be room in local store for some of these registers, but the increased speed of access is not worth the extra decoding required to get at these register values when they are used in address modification. The addresses of the registers in the first 16 locations in control store are identical to their register numbers as specified by the Td field in the instruction or indirect word. The order is as follows:

Address	Modification	
0	N	always contains zero for no modification
1	AU	
2	QU	
3	DU	not used, since DU is decoded separately
4	IC	copy of PPR.IC
5	AL	
6	QL	
7	DL	not used for same reason as DU
8-15	0-7	index registers (X0-X7)

Each time the instruction counter PPR.IC is updated, it must be copied into control store location 4. All other registers are permanently and uniquely assigned in the locations specified. Note that the two halves of A and Q are not next to each other. This is not seen to cause too much difficulty, since the two halves have to be referenced separately anyway.

The PRs are stored in the next 32 locations. Each field in the PR is stored in a separate control store location for ease of access. The locations are as follows:

16-23	PRn.RN
24-31	PRn.SEGNO

32-39	PRn.WORDNO
40-47	PRn.BITNO

The control store address of the high 3-bit field of a particular PR is 16+n, where n is the PR number. Reference to the 4 fields in a particular PR is accomplished by incrementing the control store address by 8 for each successive field.

Following the PRs, beginning at control store location 48, the table containing information about each opcode is stored in numerical order of opcode. The leftmost 6 bits of each entry contain flags and the rightmost 12 bits point to the nanoprogram that executes the instruction.

The usual method of building an emulator on the QM-1 is to completely nanocode important key routines and instructions. After that a useful micro-language can be defined for the emulation. In this way the loss of parallelism introduced by the relatively vertical structure of a microprogram has a minimal effect on execution time of the most important or commonly used instructions. Therefore this benchmark emulation is not concerned with the nature or definition of the microinstructions to be used. The sample address preparation and execution of the store A instruction discussed in the beginning of this section are entirely programmed in nanocode, and the assumption is that no significant time will be lost when a later conversion is made to microcode. It may actually turn out that the entire address preparation will be specified by one microinstruction.

A sample nanoprogram flowchart of the store A instruction with its specified modifications is shown in Figure 29. Before discussing this flowchart further, a few definitions of notation will be useful.

Rn	Specifies a local store register n = 0, 1, 2, ..., 31.

CS(Rn)	Contents of control store location pointed to by Rn.

n ⟶ RMI	Specifies one of four functions (n=0-3) performed by the RMI unit when data is next stored from the RMI unit into local store:

0 No change in data, transfer all 18 bits to destination.

1 Bits 0-9 of source are put into bits 8-17 of destination, bits 0-7 of destination are zeroed.

2 Bits 12 and 13 of the source (Tm field) are put into the destination right adjusted.

3 Bits 14-17 of the source (Td field) are stored in the destination right adjusted.

C, A, B These are names of the three 6-bit fields in R31, left to right, respectively.

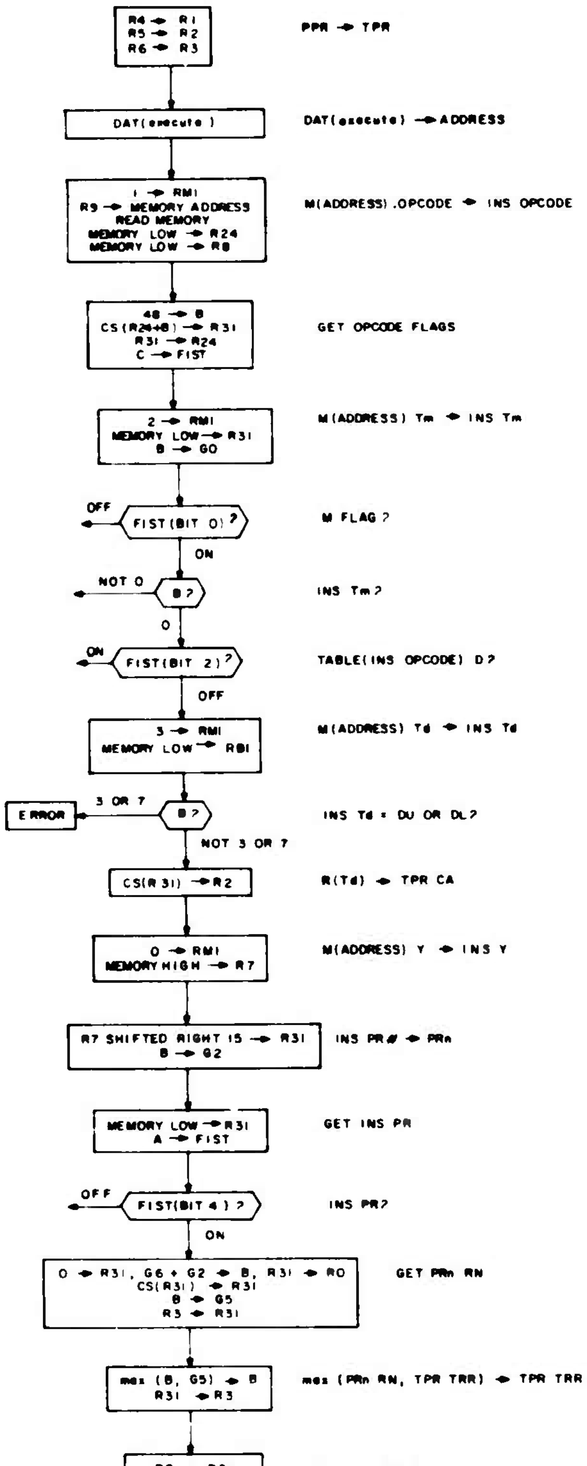
FIST The name of an F register whose bits can be individually tested.

NPC Nanoprogram counter

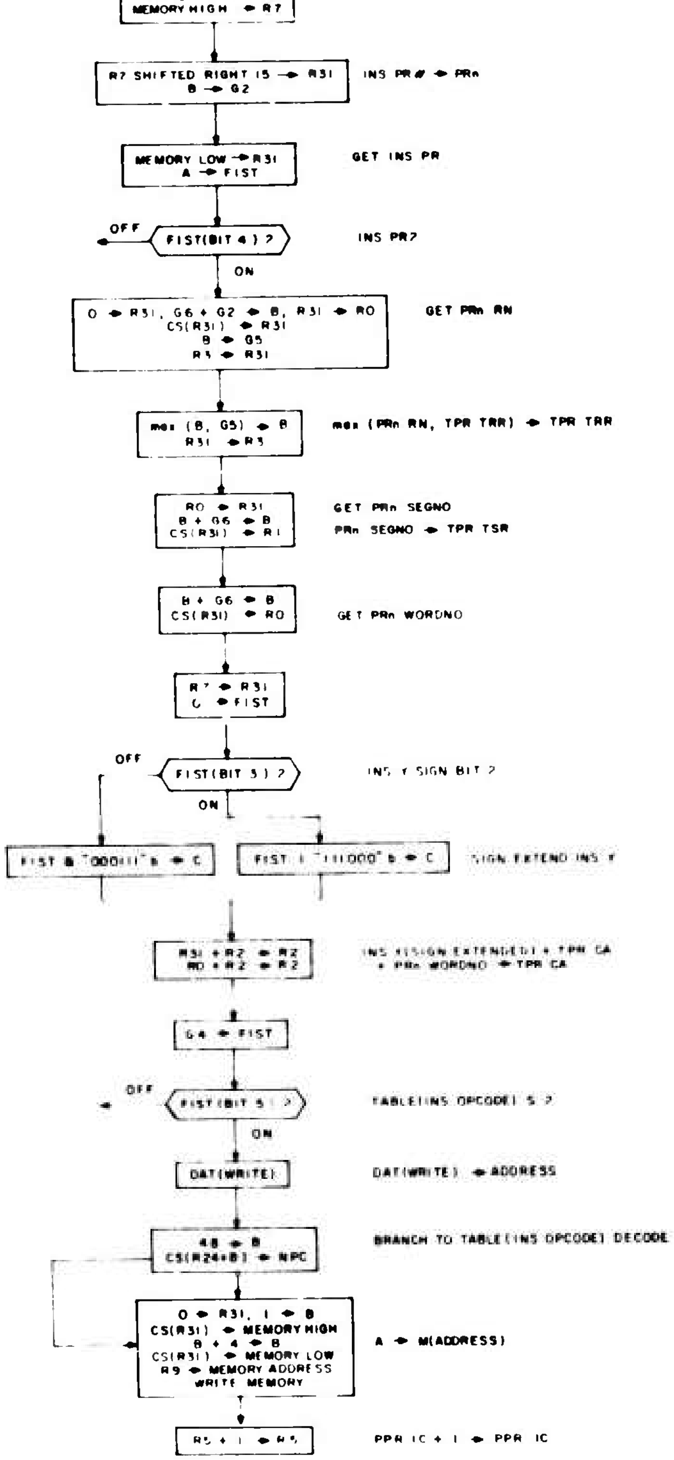
All other operations specified in the flowchart should be self-explanatory.

The flowchart in Figure 29 is at a relatively high level. Individual operations in the flowchart correspond to one or more nanoprimitives. Some operations require more than one T-step, others require less than one. For example, the operation R4→R1 requires 2 nanoprimitive functions for setting up the bus control (using constants in a K field) and another nanoprimitive for gating the transfer of data. It is not completely possible to count the number of nanoprimitives or T-steps required by this flowchart without actually doing the nanocoding. Nanocoding requires skill in determining what functions can or should be combined in T-steps and a bit of cleverness in restructuring the flow so that the number of T-steps is minimized. One reason for our not attempting to nanocode this any further is that the skilled nanoprogrammer must be the one who defines the data structures in control store and local store. It is very unlikely that the structures defined above are the best possible.

A rough estimate of the number of T-steps required to execute the flowchart is 60. This figure was arrived at assuming the maximum number of steps were required to perform each box in the flowchart with no overlap or simultaneous transfers taking place. At 80 nanoseconds per T-step, this comes out to 4.8 microseconds in the worst case. To be added to this figure should be the time required to read memory once and part of the time required to write memory. In addition, if the DAT box takes any additional time to compute an


R4 → R1
R5 → R2
R6 → R3
PPR → TPR
DAT(execute)
DAT(execute) → ADDRESS
1 → RM1
R9 → MEMORY ADDRESS
READ MEMORY
MEMORY LOW → R24
MEMORY LOW → R8
M(ADDRESS).OPCODE → INS OPCODE
48 → B
CS(R24+B) → R31
R31 → R24
C → FIST
GET OPCODE FLAGS
2 → RM1
MEMORY LOW → R31
B → G0
M(ADDRESS) Tm → INS Tm
OFF
FIST(BIT 0)?
M FLAG ?
ON
NOT 0
B?
INS Tm?
0
ON
FIST(BIT 2)?
TABLE(INS OPCODE) D?
OFF
3 → RM1
MEMORY LOW → R81
M(ADDRESS) Td → INS Td
ERROR
3 OR 7
B?
INS Td = DU OR DL?
NOT 3 OR 7
CS(R31) → R2
R(Td) → TPR CA
0 → RM1
MEMORY HIGH → R7
M(ADDRESS) Y → INS Y
R7 SHIFTED RIGHT 15 → R31
B → G2
INS PR# → PRn
MEMORY LOW → R31
A → FIST
GET INS PR
OFF
FIST(BIT 4)?
INS PR?
ON
0 → R31, G6 + G2 → B, R31 → R0
CS(R31) → R31
B → G5
R3 → R31
GET PRn RN
max (B, G5) → B
R31 → R3
max (PRn RN, TPR TRR) → TPR TRR

6 2



ID-41,910

Figure 29. QM-1 Emulation Flowchart

address, two calls to DAT should be added in. It seems reasonable then to assume that the store instruction could be executed in 6-10 microseconds as compared to the 6180 time of 1 microsecond. This figure is consistent with initial estimates made by Nanodata personnel when presented with a description of the 6180.

THE BURROUGHS B1700

Our understanding of the B1700 is based almost entirely on the contents of a preliminary edition of the Burroughs B1700 Systems Reference Manual [15] . This manual is somewhat incomplete as it does not document all of the microinstructions, provide any timing information, or explicity specify the data paths in the CPU. Crude timing information is available from other sources [16] . We were unable to talk to any technical people within Burroughs to increase our understanding of the B1700.

Description of the B1700

The B1700 is a family of small to medium scale computer systems designed to compete with the IBM System/3 family and the very low end of the IBM 360/370 series. The preliminary Systems Reference Manual describes three models, the B1712, B1714, and B1726, and recently (July, 1973) a fourth model, the B1728 has been announced. The B1728 is the largest member of the family and we will use a combination of B1728 and B1726 characteristics in our feasibility study. Main memory for the B1700 uses LSI MOS technology, and has a read access time of 180 nanoseconds and a full cycle time of about 700 nanoseconds. The B1726 can have up to 98,304 bytes of main memory, the B1728 can have 262,144 bytes. Both the B1726 and B1728's central processors operate at 6 million cycles per second, and both have control memory as well as main memory. The control memory on the B1726 has a maximum capacity of 2048 microinstructions. The B1728 can have control memory for 4096 microinstructions.

The single most unique feature of the B1700 as a microprocessor is that it represents a first step toward eliminating certain inherent structural components that are found in other microprocessors, and in fact, in most computers. A close examination of most microprocessors will reveal that the emulation of certain host architecture/instruction sets is much more feasible than that of others.

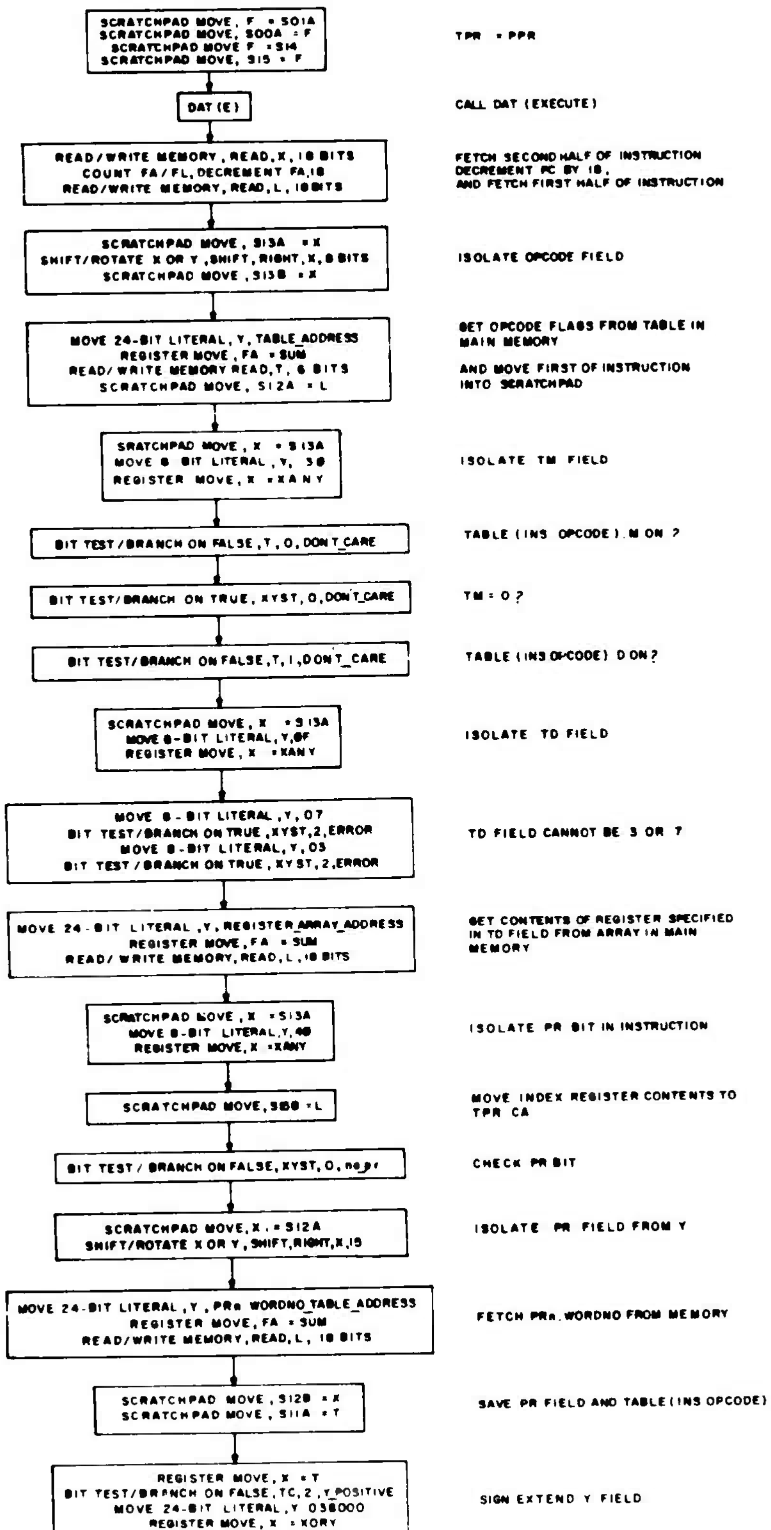
The Digital Scientific Corporations META 4, for example, can be used to reasonably emulate an IBM S/360, but not a DEC PDP-11 [17]. One way in which the B1700 gives the appearance of no inherent structure is that there are no word sizes or data formats - operands may be in any shape or size without loss of efficiency.

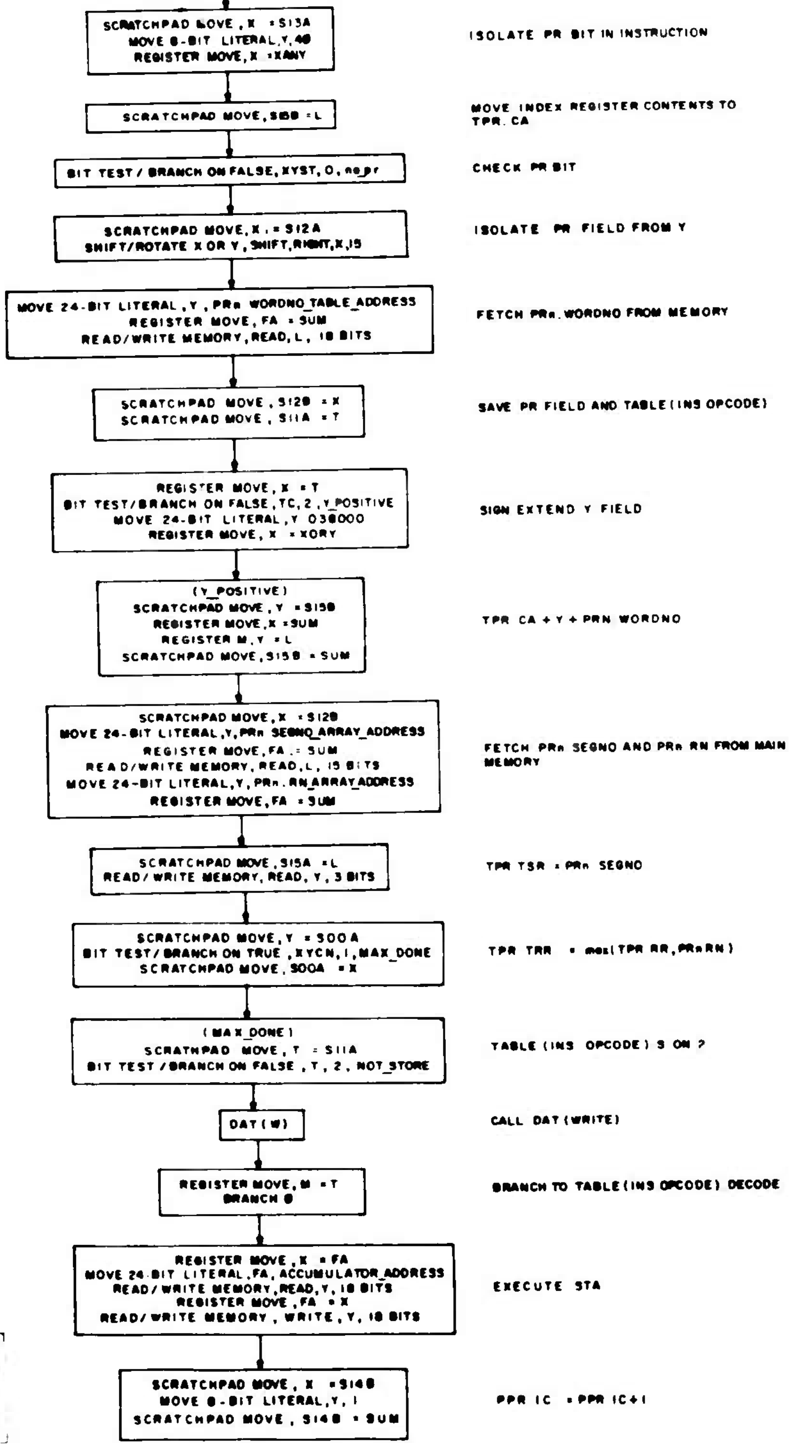
While the internal data paths of the CPU are 24 bits wide, main memory is bit addressable and can be read or written in quantities of 1 to 24 bits. The B1700 has only 4 general purpose registers, a memory address register, and a scratchpad that can be accessed as 32 24-bit words or 16 48-bit words. In addition, a 32 word stack is provided, allowing the nesting of micro subroutines. Hardware condition bits are held in a set of 4-bit registers and the 24-bit registers can be mapped into 4-bit fields. Two of the general purpose registers are continuously used as inputs to a 24-bit function box. The results of various operations on one or two variables (sum, difference, logical and, complement, etc.) are selected by using the appropriate pseudo register in the source field of a microinstruction.

Honeywell 6180 Emulation

In emulating the 6180 we will use the standard B1700 hardware as described in documentation available to us, except we will assume that appropriate dynamic address translation (DAT) hardware can be interfaced with the CPU. In doing an emulation the first task is to map the data structures of the emulated (or target) machine onto the host machine. In this emulation it is necessary to keep all programmer accessible registers (pointer registers, index registers, accummulator, and quotient register) in main memory, as well as the table used for decoding 6180 instructions, because there is no other place for them. The scratchpad is too small and control memory can only be used to hold microinstructions. Low level 6180 registers such as the TPR and PPR and variables internal to the emulation are kept in the scratchpad.

The benchmark emulation of the STA instruction (see Figure 30) requires 76 microinstructions. Assuming 250 nanoseconds (about 1½ machine cycles) per microinstruction, this is an execution time of 19 microseconds. Delays due to main memory accesses could easily increase this time to 25 microseconds. Several other factors diminish the feasibility of emulating a 6180 with a B1700. The maximum size of control memory on the B1728 is 4096 microinstructions. It is highly unlikely that this is large enough for a full emulation - thus either microinstructions would have to be executed out of main


SCRATCHPAD MOVE, F = S01A
SCRATCHPAD MOVE, S00A = F
SCRATCHPAD MOVE F = S14
SCRATCHPAD MOVE, S15 = F
TPR = PPR
DAT (E)
CALL DAT (EXECUTE)
READ/WRITE MEMORY, READ, X, 18 BITS
COUNT FA/FL, DECREMENT FA,18
READ/WRITE MEMORY, READ, L, 18BITS
FETCH SECOND HALF OF INSTRUCTION
DECREMENT PC BY 18,
AND FETCH FIRST HALF OF INSTRUCTION
SCRATCHPAD MOVE, S13A = X
SHIFT/ROTATE X OR Y, SHIFT, RIGHT, X, 6 BITS
SCRATCHPAD MOVE, S13B = X
ISOLATE OPCODE FIELD
MOVE 24-BIT LITERAL, Y, TABLE_ADDRESS
REGISTER MOVE, FA = SUM
READ/WRITE MEMORY READ, T, 6 BITS
SCRATCHPAD MOVE, S12A = L
GET OPCODE FLAGS FROM TABLE IN MAIN MEMORY
AND MOVE FIRST OF INSTRUCTION INTO SCRATCHPAD
SRATCHPAD MOVE, X = S13A
MOVE 8 BIT LITERAL, Y, 30
REGISTER MOVE, X = XANY
ISOLATE TM FIELD
BIT TEST/BRANCH ON FALSE, T, 0, DONT_CARE
TABLE (INS OPCODE).M ON ?
BIT TEST/BRANCH ON TRUE, XYST, 0, DON'T_CARE
TM = 0 ?
BIT TEST/BRANCH ON FALSE, T, 1, DONT_CARE
TABLE (INS OPCODE) D ON ?
SCRATCHPAD MOVE, X = S13A
MOVE 8-BIT LITERAL, Y, 0F
REGISTER MOVE, X = XANY
ISOLATE TD FIELD
MOVE 8-BIT LITERAL, Y, 07
BIT TEST/BRANCH ON TRUE, XYST, 2, ERROR
MOVE 8-BIT LITERAL, Y, 03
BIT TEST/BRANCH ON TRUE, XYST, 2, ERROR
TD FIELD CANNOT BE 3 OR 7
MOVE 24-BIT LITERAL, Y, REGISTER_ARRAY_ADDRESS
REGISTER MOVE, FA = SUM
READ/WRITE MEMORY, READ, L, 18 BITS
GET CONTENTS OF REGISTER SPECIFIED IN TD FIELD FROM ARRAY IN MAIN MEMORY
SCRATCHPAD MOVE, X = S13A
MOVE 8-BIT LITERAL, Y, 40
REGISTER MOVE, X = XANY
ISOLATE PR BIT IN INSTRUCTION
SCRATCHPAD MOVE, S8B = L
MOVE INDEX REGISTER CONTENTS TO TPR CA
BIT TEST/BRANCH ON FALSE, XYST, 0, no_pr
CHECK PR BIT
SCRATCHPAD MOVE, X = S12A
SHIFT/ROTATE X OR Y, SHIFT, RIGHT, X, 15
ISOLATE PR FIELD FROM Y
MOVE 24-BIT LITERAL, Y, PRn WORDNO_TABLE_ADDRESS
REGISTER MOVE, FA = SUM
READ/WRITE MEMORY, READ, L, 18 BITS
FETCH PRn.WORDNO FROM MEMORY
SCRATCHPAD MOVE, S12B = X
SCRATCHPAD MOVE, S11A = T
SAVE PR FIELD AND TABLE (INS OPCODE)
REGISTER MOVE, X = T
BIT TEST/BRANCH ON FALSE, TC, 2, Y_POSITIVE
MOVE 24-BIT LITERAL, Y 038000
REGISTER MOVE, X = XORY
SIGN EXTEND Y FIELD


SCRATCHPAD MOVE, X = S13A
MOVE 8-BIT LITERAL, Y, 40
REGISTER MOVE, X = XANY
ISOLATE PR BIT IN INSTRUCTION
SCRATCHPAD MOVE, S0B = L
MOVE INDEX REGISTER CONTENTS TO TPR. CA
BIT TEST / BRANCH ON FALSE, XYST, 0, no_pr
CHECK PR BIT
SCRATCHPAD MOVE, X . = S12A
SHIFT/ROTATE X OR Y, SHIFT, RIGHT, X, 15
ISOLATE PR FIELD FROM Y
MOVE 24-BIT LITERAL, Y, PRn WORDNO_TABLE_ADDRESS
REGISTER MOVE, FA = SUM
READ/WRITE MEMORY, READ, L, 18 BITS
FETCH PRn. WORDNO FROM MEMORY
SCRATCHPAD MOVE, S12B = X
SCRATCHPAD MOVE, S11A = T
SAVE PR FIELD AND TABLE (INS OPCODE)
REGISTER MOVE, X = T
BIT TEST/BRANCH ON FALSE, TC, 2, Y_POSITIVE
MOVE 24-BIT LITERAL, Y 038000
REGISTER MOVE, X = XORY
SIGN EXTEND Y FIELD
(Y_POSITIVE)
SCRATCHPAD MOVE, Y = S15B
REGISTER MOVE, X = SUM
REGISTER M, Y = L
SCRATCHPAD MOVE, S15B = SUM
TPR CA + Y + PRN WORDNO
SCRATCHPAD MOVE, X = S12B
MOVE 24-BIT LITERAL, Y, PRn SEGNO_ARRAY_ADDRESS
REGISTER MOVE, FA .= SUM
READ/WRITE MEMORY, READ, L, 15 BITS
MOVE 24-BIT LITERAL, Y, PRn. RN_ARRAY_ADDRESS
REGISTER MOVE, FA = SUM
FETCH PRn SEGNO AND PRn RN FROM MAIN MEMORY
SCRATCHPAD MOVE, S15A = L
READ/WRITE MEMORY, READ, Y, 3 BITS
TPR TSR = PRn SEGNO
SCRATCHPAD MOVE, Y = S00A
BIT TEST/BRANCH ON TRUE, XYCN, 1, MAX_DONE
SCRATCHPAD MOVE, S00A = X
TPR TRR = max(TPR RR, PRn RN)
(MAX_DONE)
SCRATHPAD MOVE, T = S11A
BIT TEST/BRANCH ON FALSE, T, 2, NOT_STORE
TABLE (INS OPCODE) S ON ?
DAT (W)
CALL DAT (WRITE)
REGISTER MOVE, M = T
BRANCH 0
BRANCH TO TABLE (INS OPCODE) DECODE
REGISTER MOVE, X = FA
MOVE 24-BIT LITERAL, FA, ACCUMULATOR_ADDRESS
READ/WRITE MEMORY, READ, Y, 18 BITS
REGISTER MOVE, FA = X
READ/WRITE MEMORY, WRITE, Y, 18 BITS
EXECUTE STA
SCRATCHPAD MOVE, X = S14B
MOVE 8-BIT LITERAL, Y, 1
SCRATCHPAD MOVE, S14B = SUM
PPR IC = PPR IC+1

[ID-41,911]

Figure 30. B1700 Emulation

memory (which the B1700 can do) or micro routines would have to be "demand paged" into control memory. In either case performance would suffer. Also, the maximum size of main memory - 256K bytes for a B1728 - is very small compared to a "typical" 6180 configuration. Even the theoretical limit of 2^{21} bytes of main memory (based on the size of the memory address register) is on the small side.

SECTION V

SUMMARY

The Honeywell 6180 is a large and powerful machine, containing many features not found on other large-scale computers. The complex addressing structure, virtual memory, and extended instruction set make emulation a difficult task. In addition, lack of complete documentation at this point leaves uncertainties that must be taken care of if an emulation attempt is to proceed. We have drawn up "best guess" flowcharts of the instruction cycle, address preparation, and dynamic address translation for the purpose of defining the machine in enough detail so that a benchmark emulation can be performed. In addition, a general method for complete emulation of the 6180 has been described.

The goal of the benchmark emulation was to determine which of three machines -- the D-Machine, the QM-1, and the B1700 -- are suited for emulation of the 6180, and whether such an emulation is feasible. We have concluded that the D-Machine and the QM-1 can handle the emulation, as they are general purpose microprogrammable machines, but the B1700, with its small size and relatively limited computational power, probably could not. The sample benchmark emulated a store A instruction using two of the common addressing modes. This sample gives a rough estimate of the relative processor speeds in the general case. The relative speeds may not necessarily be the same for the EIS instructions, which are much more complex in their execution cycles, nor for interrupts or dynamic address translation involving memory fetches. We are confident, however, that our example gives us a good idea of the overall time that will be used within the CPU to execute an "average" program.

It should be noted that the response time seen by the operator at a Multics terminal is as much a function of load, memory size, speed of I/O, and other variables of the application or installation, as it is of processor speed. For example, a 6180 with a small amount of memory and heavy load resulting in frequent paging may appear to have a poorer performance than an emulator that runs at a fraction of the speed but has more memory. The choice of emulator should therefore be based on the expected use and load on the machine -- a decision that can not be made here. In addition, reliability and support by the manufacturer of the emulator must be considered.

Unfortunately, it was not possible within the scope of this project to actually microcode the benchmark emulation on all three machines. We believe, however, that our calculations are fairly accurate, and they have been mostly supported by similar estimates made by people experienced in microcoding of the machines. For each of the three machines, the processor speed reduction has been calculated as follows:

D-Machine	11-18 times
QM-1	6-10 times
B1700	25 times

Edmund L. Burke
Management and Computer Systems

Morrie Gasser

Morrie Gasser
Management and Computer Systems

W. Lee Schiller

W. Lee Schiller
Management and Computer Systems

REFERENCES

1. Schroder, Michael D., and J. H. Saltzci, "Hardware Architecture for Implementing Protection Rings", Communications of the ACM, (New York: Association for Computing Machinery, March 1972) XV, 3.

2. "Preliminary Multics System Summary Description", (HIS, Inc., April 1973) DSO-73-3-4.

3. Andrews, J., M. L. Goudy, J. E. Barnes, Model 645 Processor Reference Manual, Cambridge Information Systems Laboratory, AH82 (HIS, Inc., April 1, 1971).

4. Summary of the H6180 Processor, Supplement to the 645 Processor Manual (MIT Information Processing Center, May 22, 1973).

5. "Series 6000 Features for the Multics Virtual Memory", Multics Technical Papers, (HIS, Inc., April 1972).

6. Series 6000 Summary Description, (HIS, Inc., 1971).

7. Macro Assembler Program (GMAP) (Series 600/6000 Software, BN86), (HIS, Inc., October 1972).

8. Extended Instruction Set Processor (Series 6000 Hardware, DB27), (HIS, Inc., October 1972).

9. QM-1 Hardware Level User's Manual, (Nanodata Corporation, March 21, 1973).

10. Bensoussan, A., C. T. Clingen, R. C. Daley, "The Multics Virtual Memory", Second Symposium on Operating System Principles, Princeton University, October, 1969, (New York: Association for Computing Machinery, 1969), pg. 30-42.

11. Reigel et al, "The Interpreter-A Microprogrammable Processor", AFIPS Conf. Proc. (SJCC) Vol. 40, (Montvale NJ: AFIPS Press, 1972).

12. Bingham et al, "Microprogramming Manual for Interpreter Based Systems", (Burroughs Corp. TR 70-8 November, 1970).

13. Davis et al, "A Building Block Approach to Multiprocessing", AFIPS Conf. Proc. (SJCC) Vol. 40, (Montvale NJ: AFIPS Press, 1972).

14. Organick, Elliott I., The Multics System: An Examination of Its Structure, (Cambridge, Mass: The MIT Press, 1972).

15. Burroughs B1700 Systems Reference Manual (Preliminary Edition), (Burroughs Corporation, April 1972).

16. Wilner, W. T., "Design of the Burroughs B1700", AFIPS Conference Proceedings, Vol. 41, (Montvale, NJ: AFIPS Press, 1972).

17. Anagnostopoulos, P., M. J. Michel, G. H. Sockut, G. M. Stabler, A. van Dam, "Computer Architecture and Instruction Set Design", AFIPS Conference Proceedings, Vol. 42 (Montvale, NJ: AFIPS Press, 1973).